National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Concurrent Protocol Data Unit Encoding / Decoding: Algorithms, Architectures and Performance Evaluation

Izzet Murat Bilgic

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirement

for the Degree of Doctor of Philosophy at

Concordia University

Montreal, Quebec, Canada

June, 1992

ISBN 0-315-80920-5

Canada

CONCORDIA UNIVERSITY
Division of Graduate Studies

This is to certify that the thesis prepared

By: Murat Bilgic

Entitled: Concurrent Protocol Data Unit Encoding/Decoding: Algorithms, Architectures and Performance Evaluation

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. M. Kusy — Chair

Dr. A.S. Krishnakumar — External Examiner

Dr. H.F. Li — Concordia External to Program

Dr. A.J. Al-Khalili — Internal Examiner

Dr. B.S. Haroun — Internal Examiner

Dr. K. Thulasiraman — Internal Examiner

Dr. B. Sarikaya — Thesis Supervisor

Approved by M.O. Ahmad
Dr. M.O. Ahmad
Graduate Program Director
Department of Electrical and Computer Engineering

July 13 1992

Dr. M.N.S. Swamy
Dean, Faculty of Engineering
and Computer Science

ABSTRACT

Concurrent Protocol Data Unit Encoding / Decoding:

Algorithms, Architectures and Performance Evaluation

Izzet Murat Bilgic, Ph.D

Concordia University, 1992

This thesis entails the development of concurrent algorithms and specific architectures for the Protocol Data Unit (PDU) encoding / decoding functionality of communication protocol implementations. The main motivation in the development of these algorithms and architectures is to increase the end-to-end performance of multilayered communication architectures where high computational costs associated with PDU encoding / decoding directly affect the performance. In order to construct the formal basis of this development, the notions of type and value of PDUs are initially defined. Based on the type-value duality and different forms of values, PDU encoding and decoding are divided into phases, which are then identified as mappings between different value forms. Founded on this formalism, concurrent algorithms for PDU encoding / decoding are developed. A distributed implementation model for these algorithms is then conceptualized, where the Communicating Sequential Processes (CSP) notation is used as the basis of formalism to specify the model. Using the distributed implementation model, a taxonomy for different architectural models for PDU encoders / decoders is presented. A customized performance evaluation methodology for measuring the effectiveness of the architectures is proposed. The overall applicability of all the above concepts is demonstrated through the development, implementation, and performance evaluation of an Abstract Syntax Notation-One (ASN.1) encoder / decoder used in Open System Interconnection (OSI) implementations.

ACKNOWLEDGEMENTS

Foremost, I would like to express my gratitude to Professor B. Sarikaya, for his guidance, help, suggestions, positive and negative criticisms, as well as his continuous support during different phases of our research and writing.

I would also like to express my sincerest thanks to Professor G.v. Bochmann, Idacom-CRSNG-CCRIT Industrial Chair in Communication Protocols for his financial support as well as his assistance in enabling me to use the University of Montreal's facilities.

The contributions made by CRIM in the forms of scholarships were much appreciated and signified a substantial portion of my ability to continue this thesis.

On the personal side, I would like to thank my wife, Meltem, who has given me a tremendous amount of support from the first day that I started the Ph.D. program. I sincerely doubt that I would have been able to get through it all without her by my side. Last but not least, I would like to acknowledge the moral support given to me by my parents and in-laws across the miles.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

ACSE	Association Control Service Element
ARPANET	Advanced Research Projects Agency Network
ASE	Application Service Element
ASN.1	Abstract Syntax Notation One
ATRG	Abstract Transition-Relation Graph
BER	Basic Encoding Rules
CASN1	ASN.1-C Compiler
CC	Central Controller
CFG	Context-Free Grammar
CFSM	Communicating Finite State Machine
CSP	Communicating Sequential Processes
DER	Distinguished Encoding Rules
DI	Direct Interest
DTRG	Detailed Transition-Relation Graph
ED	Encoder / Decoder
EI	Entity-Initiator
ER	Entity-Responder
EFSM	Extended Finite State Machine
EU	Execution Unit
FCFS	First-Come First-Served
FIFO	First-In First-Out
FSM	Finite State Machine
FTAM	File Transfer Access Management
HDLC	High-Level Data Link Control
IC	Interface Controller
ICI	Interface Control Information
IDU	Interface Data Unit

II	Indirect Interest
IMP	IMPlementation Model
I/O	Input/Output
IP	Internet Protocol
IS	Infinite Server
ISO	International Organization for Standardization
ISODE	ISO Development Environment
LCFSPR	Last-Come First-Served Preemptive Resume
LP	Liveness Property
MA	Message Assembler
MP	Message Parser
NBS	National Bureau of Standards
OSI	Open Systems Interconnection
PCI	Protocol Control Information
PDU	Protocol Data Unit
PER	Packed Encoding Rules
PO	Processing Overhead
PS	Processor Sharing
RISC	Reduced Instruction Set Computer
ROS	Remote Operations
ROSE	Remote Operations Service Element
RPC	Remote Procedure Call
SAP	Service Access Point
SDU	Service Data Unit
SNA	Systems Network Architecture
SP	Safety Property
SPN	Stochastic Petri Nets
SW	Status Word

TCP	Transmission Control Protocol
UI	User-Initiator
UR	User-Responder
US	Underlying Service
XDR	eXternal Data Representation

LIST OF FIGURES AND TABLES

Chapter 1 INTRODUCTION

One of the most striking features of today's information age is the rapid convergence of information gathering, processing, and distribution. This rather revolutionary change is mainly due to the increasingly sophisticated *computer networks*.

A computer network is an interconnected collection of autonomous computers [Tan 88]. When two computers are able to exchange information they are said to be *interconnected*. The information exchange medium may change from copper wires to communication satellites.

A remarkable aspect of computer communication is its similarity to the communications between the species of the same kind. Like humans, dogs, birds, etc., computers engage in communication with each other using symbols and previously agreed-upon conventions. Computers are instructed to interact using symbols which are interpreted according to these conventions.

Unambigious computer communication requires computers to understand the purpose and meaning of each other's symbols. This requirement implies that computers communicate with a common set of symbols and an unambigious representation of these symbols. This feature is the basis of the use of *standards*.

Origins of today's communication standards are due to the emergence of layering and structured design techniques of computer networks and the increasing need for communication between computers from different vendors during the 1970's [Bla 91]. ARPANET [Fei 78] of Advanced Research Projects Agency of the U.S. Department of Defence and

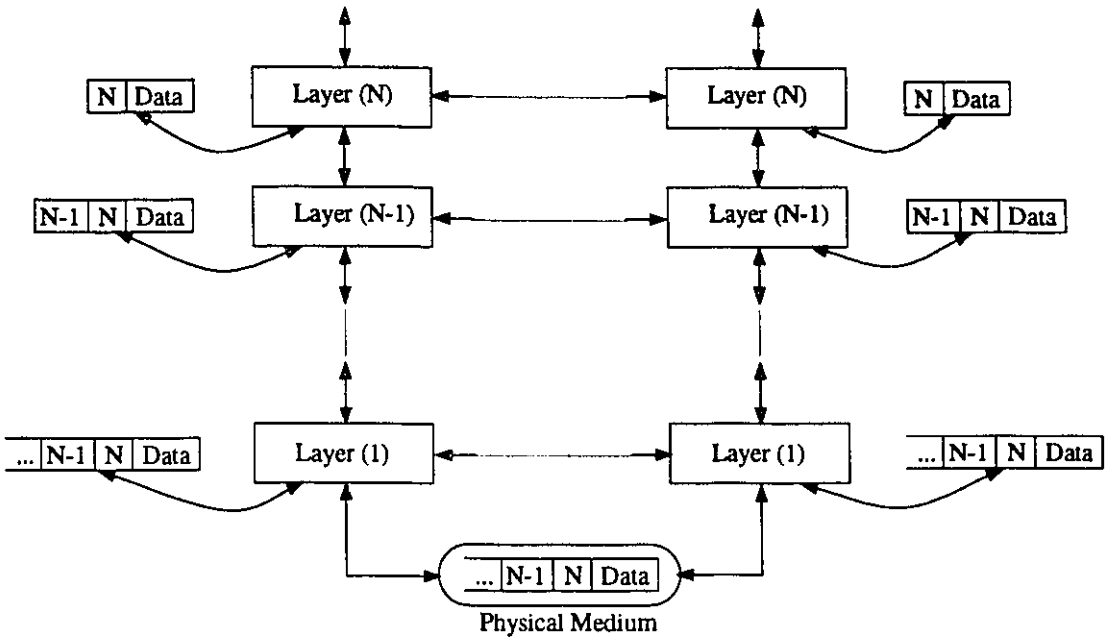
Systems Network Architecture (SNA) [Mei 87] of IBM are examples of such efforts. A more recent and completely open framework is the Open Systems Interconnection (OSI) reference model [Day 83] of the International Standards Organization (ISO).

1.1 Layered Communication Architectures

Today's increasingly sophisticated computer networks require very complex software and hardware to support a wide variety of communication activities. In order to reduce the design complexity, most networks are structured as an ordered series of *layers*. The purpose of each layer is to provide certain operations to the higher layers without any constraint on how these operations are actually implemented.

The relationship between layers is shown in Fig. 1.1. Each layer *entity* on one computer engages in a communication with the peer layer entity on another computer. This horizontal communication between peer entities is based on the layer *protocol*. A protocol is a set of procedural rules for dialogue between peers as well as the format and meaning of data units which are exchanged. Adjacent layer entities on the same computer interact through an *interface*. This vertical communication between adjacent entities is based on the underlying layer *service*. A service is a set of operations which a layer offers to the layer above it.

Figure 1.1 Layer Communications



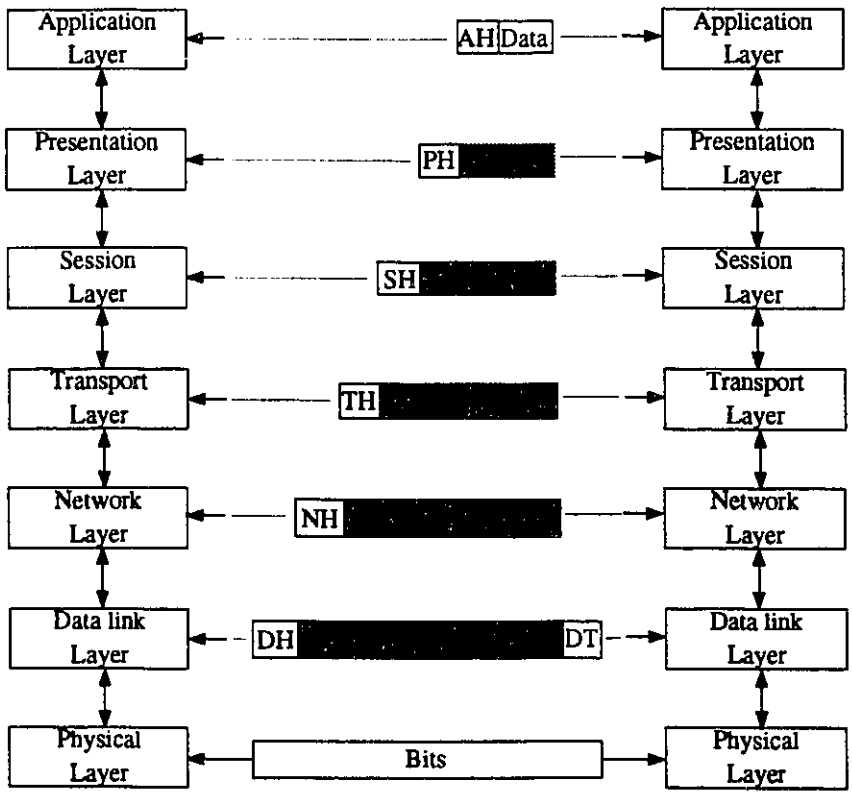
Each layer, except the lowest in most networks, adds a *header* to the data. The header includes control information which is used in the control of interaction between peer entities as well as the interaction between layer (N) entity and entities at the (N+1) and (N-1) adjacent layers. In other words, the entities of the receiving computer use the headers generated by their peers at the sending computer to interpret the accompanying data.

1.2 The OSI Reference Model

The OSI Reference Model is the best and the most comprehensive example of layered computer communication architectures [Tan 88]. It was developed to provide a common basis for communication standardization activities, guidelines for qualification of communication equipment as *open* by their adherence to standards and a common reference for standards.

The seven OSI layers are shown in Fig. 1.2. The *physical layer* is the lowest layer of the model. It is responsible for transporting bits of data from one open system to another using the physical medium which connects the two systems. The *data link layer* is responsible for delimiting the flow of bits from the physical layer as well as identifying the bits in relation to their place in a data unit. The *network layer* manages the route through which open systems direct the information to its destination. The *transport layer* is responsible for the interface between the data communications network and the upper three layers. The *session layer* coordinates the dialogue between communicating open systems. The *presentation layer* guarantees that user applications using different representations for data can communicate. The *application layer* serves as the interface between the users and the communication substructure.

Figure 1.2 Data Transmission in the OSI Model



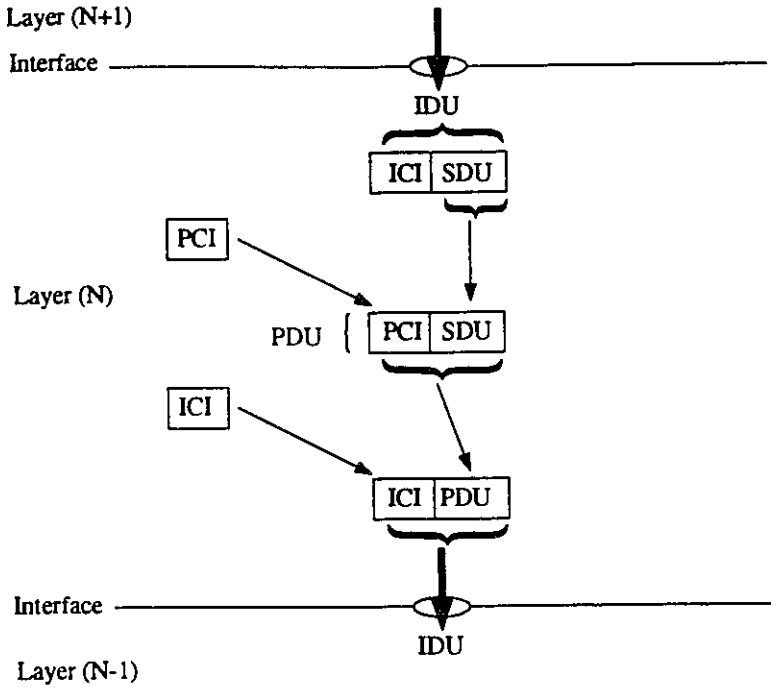
An example of data transmission using the OSI model is shown in Fig. 1.2. At the sending computer, the data of the application user are received by the application layer entity which adds its header, AH in front of the data and gives the results to the presentation layer entity which repeats the process.

In most networks, at each layer, except the physical layer, a header is added. At the data link layer, a header and a *trailer* are used to encapsulate the data coming from the network layer.

At the physical layer, bits of the data link frame consisting of the header, the trailer, and the encapsulated data are transmitted to a receiving computer. On the receiving computer, the headers are removed at the corresponding layers one by one until the original data reaches the receiving user of the application layer.

In OSI terminology, there are five components involved in the exchange of information between two layers, as shown in Fig. 1.3. At the interface between the layer (N+1) entity and the layer (N) entity, the layer (N+1) entity passes an *Interface Data Unit (IDU)* to the layer (N) entity. The IDU consists of an (N) *Service Data Unit (SDU)* complemented with *Interface Control Information (ICI)* which is used to invoke certain functions at layer (N). *Protocol Control Information (PCI)* (previously known as header) and SDU are combined to obtain (N) *Protocol Data Unit (PDU)* based on different mapping strategies to be explained later in this section. (N) PDUs combined with ICI are sent as IDUs to the layer (N-1) entity.

Figure 1.3 Components of OSI Layer



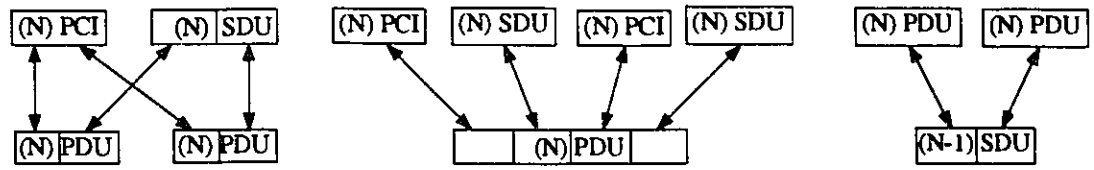
Data units in different layers are of different sizes. When the simplest case is considered, an (N) SDU fits into the user data field of an (N) PDU which is mapped to one (N-1) SDU. The OSI model also prescribes three more complex cases :

– Segmenting and reassembling : mapping of an (N) SDU into multiple (N) PDUs (Figure 1.4.a),

– Blocking and deblocking : mapping of multiple (N) SDUs into a single (N) PDU (Figure 1.4.b),

– Concatenation and separation : mapping of multiple (N) PDUs into a single (N-1) SDU (Figure 1.4.c).

Figure 1.4 Techniques for Mappings between PDUs and SDUs



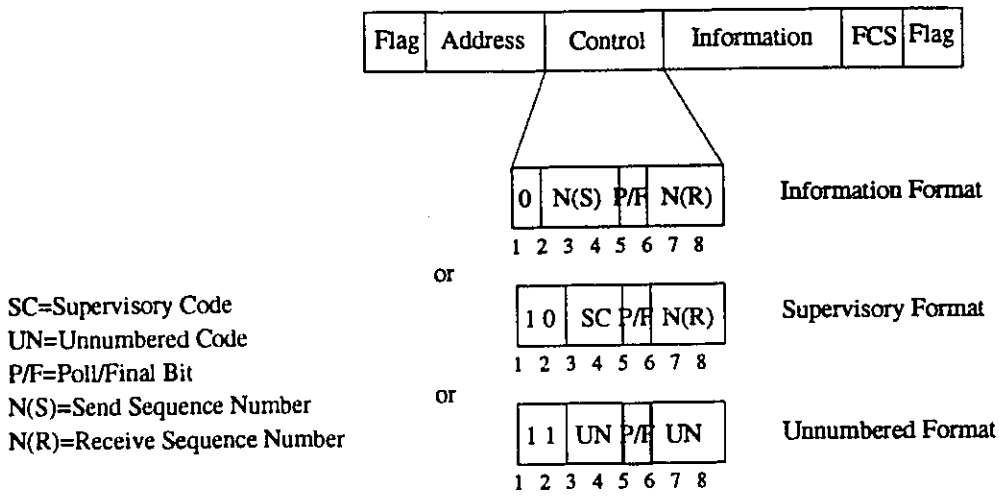
(a) Segmenting and Reassembling (b) Blocking and Deblocking (c) Concatenation and Separation

1.3 Data Unit Processing

The data which are transmitted through a communications channel between computers are identified by several names in various protocols of different reference models : *message*, *packet*, *frame*, *block*, *record*, *protocol data unit*, or *data unit*. The OSI model uses the term PDU as a general term and in this thesis we use OSI terminology to describe the data transmitted between computers.

Although the basic purpose for their use is the same for all PDUs of various layers of different reference models, the methods for representing PDUs largely differ from protocol to protocol. Almost in all communication protocols there is no provision of formalism for the representation of PDUs. Instead, they are described verbally and usually additional pictograms are used to better explain their format. Fig. 1.5 shows a typical example, High-Level Data Link Control (HDLC) frame [ISO 3309].

Figure 1.5 HDLC Frame



Although the informal descriptions and pictograms are seen descriptive enough for lower layer PDUs by the standards, they are found to be inadequate for representing complex data structures of user applications. Another difficulty in representing user application data is the heterogeneous environments in which the data are processed, and stored. Heterogeneity is due to the use of different types of computers, in turn different types of machine representations; different programming languages; and operating systems on the nodes of a computer network. OSI incorporates the use of *Abstract Syntax Notation One* (ASN.1) [ISO 8824] for the presentation and application layer PDUs to cope with the issues resulting from the heterogeneity. ASN.1 is a data declaration language used to define the complex data structures of user applications in a structural way. Fig. 1.6 shows an example ASN.1 definition for an imaginary PDU. ASN.1 treats the machine representation of data structures as an internal issue. Instead, it uses standard rules, such as *Basic Encoding Rules* (BER) [ISO 8825] to convert the values of data structures into canonical forms, a stream of octets which are understood by communicating entities. Both ASN.1 and different encoding rules are discussed in detail in chapter 6.

Figure 1.6 An Example ASN.1 Definition

```
Example DEFINITIONS ::= BEGIN
        Imag-PDU ::= SEQUENCE {
                        header  [0] BIT STRING,
                        data    [1] OCTET STRING,
                        trailer [2] BIT STRING}
END
```

As different names are used to describe the data transmitted between computers in different protocols, different terminologies are also used to specify the processing of PDUs at the sending and receiving sites. Data unit encapsulation and decapsulation are terms due to PDUs consisting of a header, a data, and a trailer part. Composition and decomposition or synthesis and analysis of PDUs are similar terms used to describe the processing of PDUs at the sending and receiving sites of a communication link, respectively. A more general term is *PDU encoding / decoding* since it corresponds to the notion of converting the data to a common external form understood by all the sites of a network, and reconstructing the data from this common external form.

Although the computational cost and the amount of necessary software for PDU encoding / decoding change from protocol to protocol, nevertheless, PDU encoding / decoding involves relatively high computational costs and requires a large amount of software. In [Gun 89], it is shown that ASN.1 encoding and decoding has a substantial degrading effect on the end-to-end performance of different OSI applications. Similarly, in [Cla 89] it is reported that packet processing constitutes a significant proportion of the Transmission Control Protocol (TCP) processing overhead. In [Svo 89], it is stated that the software for PDU encoding / decoding can amount to more than 50% of the total protocol implementation code.

In order to overcome the high computational costs for PDU encoding / decoding, specific optimization techniques are proposed. In [Sal 85], partially filled templates are

used for encoding of TCP and Internet Protocol (IP) headers. This reduces the amount of necessary data copying. Another specific technique is the prediction of the most likely next PDU [Cla 89]. According to this method, the code to precompute most likely values in the next incoming TCP packet header for the connection is added to the implementation. Although these optimization techniques are reported to yield high improvements in the computational costs involved, they are specific for each protocol and they have to be done manually [Svo 89].

Hardware based implementation of PDU encoding and decoding which is customizable for different protocols is described as a part of the programmable protocol VLSI engine [Kri 87]. The customization depends on the so-called formal specification of the PCI structure for the particular protocol. A Message Parser (MP) and a Message Assembler (MA) use the microcode generated from the PCI structure to break and assemble the header information into and from its component tokens, respectively. However, the power of this approach is limited by the constraints on the PCI structure such as the requirement of context-free PCI structures.

1.4 Related Problems

A similar problem to PDU encoding / decoding is observed in the implementation of the remote procedure call (RPC) [Bir 84] mechanism in heterogeneous environments [Gib 87, Not 88]. RPC is a paradigm for communication between computers over a network. RPC provides a user-level mechanism across the communication network such that remote procedure calls have the same or very similar syntax and semantics within the high-level language.

In heterogeneous environments, RPC implementations must deal with issues such as computers using different conventions for the representation of data values, and

programming languages using different syntax and semantics for the representation of the same data structure. A typical approach to overcome these difficulties is similar to the use of PDUs in layered communication architectures : the use of an interface specification language which provides a common base among the programming languages in the environment. The interface specification language includes a set of data types used to construct the data structures of the programming languages used in the RPC environment. It also includes a common set of machine data types, e.g. integers, booleans, etc. to specify the data types.

A typical example of RPC implementations using the interface specification languages is the standard Xerox RPC which uses the Courier protocols [Xer 81] for data representation. Another example is the SUN RPC using the XDR data representation standard [Sun 85].

The same problem is addressed, as well, in other distributed programming languages with logically distributed address spaces. Languages based on atomic transactions, such as Argus [Lis 88] or languages based on objects, such as Emerald [Bla 87] include mechanisms for communicating abstract values, or passing arguments by using a common representation.

1.5 Objective and Motivation

All the efforts related with value passing in heterogeneous environments are focused on the representation of complex data structures in a data type language and integrating the type language into different programming languages on different types of machines. On the other hand, except for the application and presentation layer PDUs of the OSI model, PDUs of different layers of various communication reference models are always considered as message streams in standards. Protocols do not provide any mechanism

to represent these PDUs on different nodes of a communication architecture; and in turn they do not provide any method to perform the mapping between the message streams and their local representations. This is largely due to the fact that the set of semantics, e.g. the meaning of PDUs to be carried in these PDUs is limited and includes relatively simple structures. Another reason is that during the standardization activities, different layers were developed by separate working groups usually in non-overlapping time periods and sometimes on an ad-hoc basis. Therefore, protocols usually reflect different styles, perspectives, and lack a common approach to common aspects such as the representation of data units. Nevertheless, the idea of encoding message streams from local representations and decoding message streams to obtain such representations can be generalized for PDU processing at any layer. On the other hand, the vast differences between the complexity and form of syntax and semantics of PDUs from various layers of different reference models hinders the possibility of developing a common language contrary to the standards.

In this thesis, we try to balance the necessity of addressing the issue of PDU encoding / decoding in a more general way, as well as the inherent difficulty of specifying a single language or representation mechanism for PDUs. Therefore, we attempt to generalize PDU encoding / decoding in a formal way, but refrain from specifying a specific language for this purpose.

The common formalism for PDUs of different layers is based on the concepts of *type* and *value*. We also define various forms for PDU values according to their residence, e.g. *local values* for N PDUs at the N layer entities, or *global values* for N PDUs at the boundary of the N-1 layer. Based on the type-value duality and different forms of PDU values we formalize the PDU encoding and decoding.

Previously, we discussed the high computational costs involved in PDU encoding /

decoding in communication architectures. A substantially faster way of PDU encoding / decoding necessitates the development of concurrent algorithms and special-purpose architectures for the implementation of these algorithms. Such architectures should be specific enough to be optimized for the PDU encoding / decoding as well as general enough to be programmable for the use of encoding / decoding PDUs of changing or emerging protocols.

There should be a mechanism to estimate the effectiveness of these alternative implementations of communications subsystems as parts of layered communication architectures. Such a mechanism should be based on a performance evaluation methodology for a full protocol stack, since such implementations may provide service to more than one layer and affect the end-to-end performance of protocol stacks.

1.6 Original Contributions

In this thesis, the concepts of type and value for PDUs are introduced. Definitions for different forms of PDUs are given and PDU encoding / decoding is formalized as transformations between forms of PDU values. The phases for PDU encoding / decoding are identified and parallel algorithms are developed for these phases.

A common distributed implementation model is developed for the phase algorithms. The correctness of the algorithms based on this model are proven and their complexities are analyzed.

A taxonomy for different models of PDU encoding / decoding architectures using the common model as a module(s) is given. The taxonomy is based on the number, and structure of modules, as well as the number of layers and sources which they serve.

A performance evaluation methodology based on two previously proposed full protocol stack performance evaluation methodologies is formalized to apply on communication architectures involving special-purpose modules such as PDU encoders / decoders. Based on this methodology, the effects of such modules on the end-to-end performance of communication architectures are measured.

Based on the concepts developed in this thesis, a multiprocessor-based programmable ASN.1 encoder / decoder is designed, the benchmark figures for the encoder / decoder and the figures for its effect on the end-to-end performance of an OSI application are obtained.

1.7 Outline of the Thesis

In chapter 2, we introduce the type-value duality concept for PDUs. Based on this duality and different forms of PDU values, encoding and decoding of PDUs are described as transformations between forms of PDU values. The phases of encoding and decoding are explained and the algorithms for these phases are also given in chapter 2.

In chapter 3, we discuss the distributed implementation model for the phase algorithms of chapter 2. The Communicating Sequential Processes (CSP) language is used as the basis of formalism to specify the model. The final distributed model is obtained as a series of refinements based on the required features of the implementation. The safety and liveness of the algorithms are shown and the time and message complexity figures for phase algorithms are derived in chapter 3.

Different architectural models to construct PDU encoders / decoders from the common distributed model of chapter 3 are presented in chapter 4. These models are classified according to different factors, such as the number and structure of modules of models, the number of layers, and the number of sources to which they give service.

In chapter 5, we discuss a performance evaluation methodology developed to measure the effect of faster PDU encoding / decoding on the end-to-end performance of layered communication architectures.

An example for the implementation of all the concepts explained in previous chapters is introduced in chapter 6. An ASN.1 encoder / decoder architecture based on a multiprocessor implementation model is discussed in detail and the effect of the ASN.1 encoder / decoder on the end-to-end performance of a typical OSI application is presented in chapter 6.

In chapter 7, we present our conclusions and lay out the possible future work in this area.

Chapter 2 PDU ENCODING AND DECODING ALGORITHMS

The vast differences between PDUs of different layers exclude the idea of using a single data type language to represent the structure of PDUs of any layer. Such an approach [ISO 8824] is chosen for application and presentation layer protocols of the OSI model where applications require the use of complex data structures and the presentation layer is designed to provide different message streams for the same local representation according to different requirements. On the other hand, lower layer PDUs of the OSI model and PDUs of other reference models are designed to carry a constant and limited set of semantics. Therefore, no provision is included in these protocols for the local representation of PDUs. However, as we explained in the first chapter a generic model can be used to describe the PDU processing at any layer. Such a generic model is based on the use of the *type-value* duality concept as in all the other models discussed in the first chapter. The basic use of such a model is to obtain concurrent algorithms for PDU processing. Since it does not depend on any specific data type language nor is associated with any particular set of protocols, it permits the development of concurrent algorithms optimized for a particular protocol.

In this chapter we introduce the generic model for PDU encoding / decoding. In order to explain the operations related to PDUs, we first examine the type-value duality for PDUs. Then based on this duality, we introduce transformations between different forms of PDU values. In order to describe the characteristics of various types and the components of PDU values, the terminology from ASN.1 and the terminology from BER are respectively used. An in-depth introduction to both ASN.1 and BER are given in

chapter 6. In the last two sections, we give the generic PDU decoding and encoding algorithms based on the introduced formalism.

2.1 Type-Value Duality

Typically, a PDU comprises of a header together with data (if any) which come from or are to be delivered to the next higher layer in an implementation of a multilayered reference model. PDUs enable the peer entities to exchange the same *semantics* which may be mapped to different *local interpretations* in an implementation.

A PDU may include varying components, since its semantics may contain optional or default parts which may be excluded from the data sent to a peer entity. In other words, the data transferred during the dialogue may be an instance of a data structure used to define the collections of all possible instances. This forms the concept of type-value duality for PDUs.

We will use *typed, attributed trees* to represent PDU types as well as values. Each node of a tree has a *built-in* type which determines the attributes of the node. Nodes in different uses have different attributes designed for PDU encoding and decoding operations, and protocol core functions.

2.1.1 PDU Types

A PDU *type* of a protocol P specifies the structure of a collection of instances of that type and is represented by a tree. The type and attributes of nodes of the tree for a PDU type differ from protocol to protocol. The environment where processing of PDUs is carried out is another factor in determining the type and attributes of a node. Similarly, the set of available built-in types is also protocol and environment dependent. Nevertheless, we can make certain generalizations about the characteristics of PDU types.

A set of *built-in* types is used for specifying the nodes of PDU types. As in many existing data specification languages, there are two classes of built-in types : *primitive* and *constructed.* If a node of the PDU type is of a primitive built-in type, then it does not refer to any other node in the PDU type. *Integers, real numbers, booleans* are typical examples for primitive built-in types. On the other hand, nodes of constructed built-in types include references to other nodes of the PDU type. Such nodes are called *components* of the original node. *Sequences, sets* are typical examples of constructed built-in types.

The attributes of a node of a PDU type are determined by the type of the node. If a node is of constructed built-in type, i.e. it has components, then *pointers* are needed to link the node to its components. Such a linkage may be direct when pointers for all the components are included in the node. However, when the number of components becomes large such an approach becomes infeasible. Instead a fixed number of pointers are used to link the node of the structured built-in type with some of its components directly, and these components are linked to other components. A binary tree is a typical example of this type of linkage mechanism.

As explained in the following sections, algorithms for PDU processing make traversals on the PDU types starting from a *root* node. Generally traversals are done in a *top-down* fashion where component nodes are reached from the node of a constructed built-in type. However, in some cases *backtracking* may be needed. In order to facilitate this type of traversal, pointers from component nodes to their parent nodes are used.

Each node has an *identifier* attribute which uniquely describes the node. This attribute is used for comparisons between types and values during the encoding and decoding of PDUs. A component node of a constructed type may be declared as *optional,* or with a *default* value. Nodes of a PDU type may also include information about valid ranges,

constraints among the components and semantic interpretations of their instances. All this information is included in terms of attributes.

The set of all PDU types of a protocol P is called the *type set* of P, $T_P = \{t_{P,i} | i \leq \#\ of\ PDU\ types\ in\ P\}$.

2.1.2 PDU Values

A PDU *value* is an instance of the type of that PDU. It consists of instances of components which are set according to the procedural rules of protocols as well as the value of data coming from or going to the next higher layer.

The set of all PDU values of a PDU type $t_{P,i}$ of protocol P is called the *value set* of $t_{P,i}$, $V_{P,i}$. The set of all PDU values of protocol P is called the value set of P, $V_P = \bigcup_{i=1}^{|T_P|} V_{P,i}$.

The following definition is the function which maps the PDU values of a given protocol to their respective types. Obviously distinct types have distinct values.

Definition 2.1.1 : The type function for values of protocol P is defined as

$$f_{T,P} : V_P \rightarrow \{1, ..., |T_P|\} \text{ such that } f_{T,P}(v) = i \Leftrightarrow v \in V_{P,i}.$$

2.2 Forms of PDU Values

A PDU value is an abstraction since it is represented in different forms depending on its use in different places. It is transferred from one entity to another in a common global form which is later transformed back to a local form suitable for operations. PDU encoding and decoding can be defined as transformations between these forms of the same value. Before explaining the transformations, let us first introduce different forms of values.

2.2.1 Local Values

Any PDU value, which is an instance of the type of that PDU, has a local interpretation at each entity. An entity stores the local intrepretation in a form suitable for protocol core functions in its local environment. Such a form is called the *local form* of the PDU value and being similar to PDU types, it is represented by a typed, attributed tree. Nodes of a PDU value in local form (in short local value) are the instances of the nodes of the PDU type.

Nodes of local value have attributes similar to those of nodes of PDU types. An important additional attribute is the value or the pointer to the value of a primitive built-in type. Another attribute is the choice of the *encoding rule* if there are different possible rules to obtain the message stream for the PDU value. Identifiers for local value nodes must be defined as functions of identifiers for their respective type nodes.

The set of all local PDU values of a PDU type $t_{P,i}$ of protocol P is called the *local value set* of $t_{P,i}$, $L_{P,i}$. The set of all local PDU values of protocol P is called the local value set of P, $L_P = \bigcup_{i=1}^{|T_P|} L_{P,i}$.

The following definition gives the mappings from abstract values into their local forms of a PDU type, and all PDU types of a protocol P, respectively. The function $f_{L,P,i}()$ is one-to-one onto since every abstract value of a PDU type $t_{P,i}$ should have a unique local form to carry the associated semantics unambigiously. Similarly the function $f_{L,P}()$ is a one-to-one onto function to guarantee that local forms of abstract values of different PDU types are distinct. This feature is also necessary to obtain a *canonical representation* for values during the transmission.

Definition 2.2.1 : a) $f_{L,P,i}$ for PDU type $t_{P,i}$ of protocol P is a one-to-one onto function such that, $f_{L,P,i} : V_{P,i} \rightarrow L_{P,i}$ where $V_{P,i}$, and $L_{P,i}$ are value set, and local value set of type $t_{P,i}$, respectively.

b) $f_{L,P}$ for protocol P is a one-to-one onto function $f_{L,P} : V_P \rightarrow L_P$ such that $f_{L,P}(v) = f_{L,P,f_{T,P}(v)}(v)$ where V_P, and L_P are value set, and local value set of protocol P, respectively.

2.2.2 Global Values

In order to exchange the same semantics during the dialogue, different entities share a common representation, a message stream for PDU values. Such a form is called the *global form* of PDU value. A PDU value in global form (in short global value) is a stream of bits generated from a given local value according to a set of rules and transferred to the next lower layer to be sent to a peer entity.

Global values can be divided into functional units for local PDU value nodes whose encodings are included in the global value. A *functional unit* is a string of *atomic units* (bit, nibble, byte, etc.). There are three classes of functional units :

a) *Identifier* unit for a local value node.

b) *Length* unit for a local value node.

c) *Contents* unit for a local value node.

The identifier unit in a global value uniquely determines the related type node and the local value node. In this way, a canonical representation for the local values is obtained. These identifiers are used in the decoding process to distinguish the type node and in turn local value node for the related contents unit. Identifier units can be either *fixed* or *variable* size. If an identifier unit is of variable size, then it must be encoded in the

identifier unit itself.

Length unit specifies the place of related contents unit in the global value. If the length unit for a local value node is not included in the global value, then the place of its contents unit must be implicitly defined by the other functional units in the global value. Therefore, the length for a local value node is either *implicit* or *explicit*. If the length is explicit, i.e. if there is a length unit, then it may be either in *definite* or *indefinite* form. If the length of a local value node is definite, then the length unit must be the coding of the size of the related functional units, e.g. contents, contents and the length itself, etc. in terms of atomic units. On the other hand, if the length of a local value node is indefinite, then the contents unit is either enclaved by a pair of special markers or a single marker shows the end of the contents unit. When the contents unit is enclaved by markers, the first marker becomes the length unit for the local value node; whereas the marker at the end of original contents is added to the contents unit as the functional unit(s) of an extra component. When the end of contents unit is marked by a single marker, it becomes the length unit. Similar to identifier units, length units may have either fixed or variable sizes in terms of atomic units.

The contents unit for a local value node of primitive type is the representation of the built-in data according to the encoding rules of the protocol. On the other hand, contents for a local value node of constructed type is the collection of functional units of its components.

For any type node whose instance exists in the local value and is needed to be sent to a peer entity to convey the semantics, at least one of the identifier and contents functional units must exist in the global value. If a functional unit for such a component instance is not included, then it must be implicitly defined in the encoding rules and other functional components in the global value.

An important constraint about the global values is that there needs to be a non-empty set of functional units in each global value whose location is a priori known for the decoding process. Otherwise, a decoding process cannot find any initial seed unit to start locating other functional units. Usually this set contains the identifier unit for the root local value node.

As discussed in the first chapter, PDU formats of existing communication protocols constitute a wide range. In some protocols, global values may include non-adjacent functional units for the same local value node as in the case of Internet Datagram [Pos 81] where identifier and length units of the root local value node, i.e. PDU, divide the functional units for its component nodes. The order of functional units in global values also changes from protocol to protocol. According to ASN.1 and BER, this order is always identifier, length, and then contents. On the other hand, according to OSI transport layer protocol [ISO 8073], while the length, identifier, contents order is used for the root local value node, i.e. PDU, the identifier, length and contents order is used for some component nodes, e.g. variables of the variable part.

The set of all global PDU values of a PDU type $t_{P,i}$ of protocol P is called the *global value set* of $t_{P,i}$, $G_{P,i}$. The set of all global PDU values of protocol P is called the global value set of P, $G_P = \bigcup_{i=1}^{|T_P|} G_{P,i}$.

As explained above, the canonical representation mechanism guarantees that distinct types and in turn distinct values have distinct global representations for each protocol. The following definition gives the one-to-one onto functions mapping abstract values into their global forms for any PDU type $t_{P,i}$ and all PDU types of a protocol P, respectively.

Definition 2.2.2 : a) $f_{G,P,i}$ for PDU type $t_{P,i}$ of protocol P is a one-to-one onto function such that, $f_{G,P,i} : V_{P,i} \rightarrow G_{P,i}$ where $V_{P,i}$, and $G_{P,i}$ are value set, and global value set of type $t_{P,i}$, respectively.

b) $f_{G,P}$ for protocol P is a one-to-one onto function $f_{G,P} : V_P \rightarrow G_P$ such that $f_{G,P}(v) = f_{G,P,f_{T,P}(v)}(v)$ where V_P, and G_P are value set, and global value set of protocol P, respectively.

2.2.3 Intermediate Values

As explained in section 2.3, encoding and decoding are transformations between local and global forms of PDU values. During these transformations, another form of value is also used as an intermediate form of representation. Such a form is called the *intermediate form* of a PDU value, and is represented as a typed, attributed tree. A PDU value in intermediate form (in short intermediate value) has nodes corresponding to the local value nodes whose encodings are to be included in the global value.

Intermediate value nodes have their corresponding functional units in the global value as attributes. If the order and adjacency characteristics for all local value nodes of a protocol are the same as in the case of ASN.1, these attributes are the only attributes to obtain the global value. On the other hand, if these characteristics are dependent on the type of the corresponding local value node, then they also need to be included as attributes.

The set of all intermediate PDU values of a PDU type $t_{P,i}$ of protocol P is called the *intermediate value set* of $t_{P,i}$, $I_{P,i}$. The set of all intermediate PDU values of protocol P is called the intermediate value set of P, $I_P = \bigcup_{i=1}^{|T_P|} I_{P,i}$.

Similar to functions mapping abstract values into their local and global forms, functions mapping them into their intermediate forms are one-to-one onto for each PDU type as well as any protocol P.

Definition 2.2.3 : a) $f_{I,P,i}$ for PDU type $t_{P,i}$ of protocol P is a one-to-one onto function such that, $f_{I,P,i} : V_{P,i} \rightarrow I_{P,i}$ where $V_{P,i}$, and $I_{P,i}$ are value set, and intermediate value set of type $t_{P,i}$, respectively.

b) $f_{I,P}$ for protocol P is a one-to-one onto function $f_{I,P} : V_P \rightarrow I_P$ such that $f_{I,P}(v) = f_{I,P,f_{T,P}(v)}(v)$ where V_P, and I_P are value set, and intermediate value set of protocol P, respectively.

2.3 Transformations on PDUs

Different forms of PDU values are introduced in the previous section. Now, we can define the transformations between these forms.

2.3.1 Encoding and Decoding

Encoding and decoding of PDUs are to be explained as transformations between local values and global values based on their types. In order to define these transformations, a set of rules is needed.

Encoding rules for a protocol P are the rules used to convert local values in L_P into global values in G_P. The reverse transformation is based on the inverse of the same encoding rules.

The following definition gives the transformations between the local values and the global values of a PDU type $t_{P,i}$ of protocol P. Both encoding and decoding are one-to-onto functions since the abstract value corresponding to the global and local form is unique according to definitions 2.2.1–2.2.3.

Definition 2.3.1 : a) *Encoding* for PDU type $t_{P,i}$ of protocol P based on the set of encoding rules R_P is a one-to-one onto function $\mathcal{E}_{P,i} : L_{P,i} \rightarrow G_{P,i}$ such that $\mathcal{E}_{P,i}(l) = g \Leftrightarrow f^{-1}_{L,P,i}(l) = f^{-1}_{G,P,i}(g)$,

b) *Decoding* for PDU type $t_{P,i}$ of protocol P based on the set of encoding rules R_P is a one-to-one onto function $\mathcal{D}_{P,i} : G_{P,i} \rightarrow L_{P,i}$ such that $\mathcal{D}_{P,i}(g) = l \Leftrightarrow f^{-1}_{L,P,i}(l) = f^{-1}_{G,P,i}(g)$,
where $L_{P,i}$, and $G_{P,i}$ are local value set, and global value set of type $t_{P,i}$, respectively.

Definition 2.3.1 establishes the correctness criteria for encoding and decoding of PDU values of PDU type $t_{P,i}$ similar to the identities given in [Her 82] as :

1. $f^{-1}_{L,P,i}(l) = f^{-1}_{G,P,i}\left(\mathcal{E}_{P,i}(l)\right)$
2. $f^{-1}_{G,P,i}(g) = f^{-1}_{L,P,i}\left(\mathcal{D}_{P,i}(g)\right)$

where l and g are respectively of $L_{P,i}$ and $G_{P,i}$. As explained in [Her 82], identity (1) intuitively specifies that l and $\mathcal{E}_{P,i}(l)$ represent the same abstract value. Similarly, identity (2) specifies that g and $\mathcal{D}_{P,i}(g)$ represent the same abstract value.

In the previous section, it is specified that $V_{P,i}$'s of any protocol P are required to be distinct. Further, it is stated that forms of the values of P are also distinct. Based on these conditions, we can define encoding and decoding transformations for protocol P as one-to-one onto functions as follows.

Definition 2.3.2 : a) *Encoding* for protocol P based on the set of encoding rules R_P is a one–to–one onto function $E_P : L_P \rightarrow G_P$ such that $E_P(l) = \mathcal{E}_{P,f_{T,P}(f^{-1}_{L,P}(l))}(l)$,

b) *Decoding* for protocol P based on the set of encoding rules R_P is a one–to–one onto function $D_P : G_P \rightarrow L_P$ such that $D_P(g) = \mathcal{D}_{P,f_{T,P}(f^{-1}_{G,P}(g))}(g)$,
where L_P, and G_P are local value set, and global value set of protocol P, respectively.

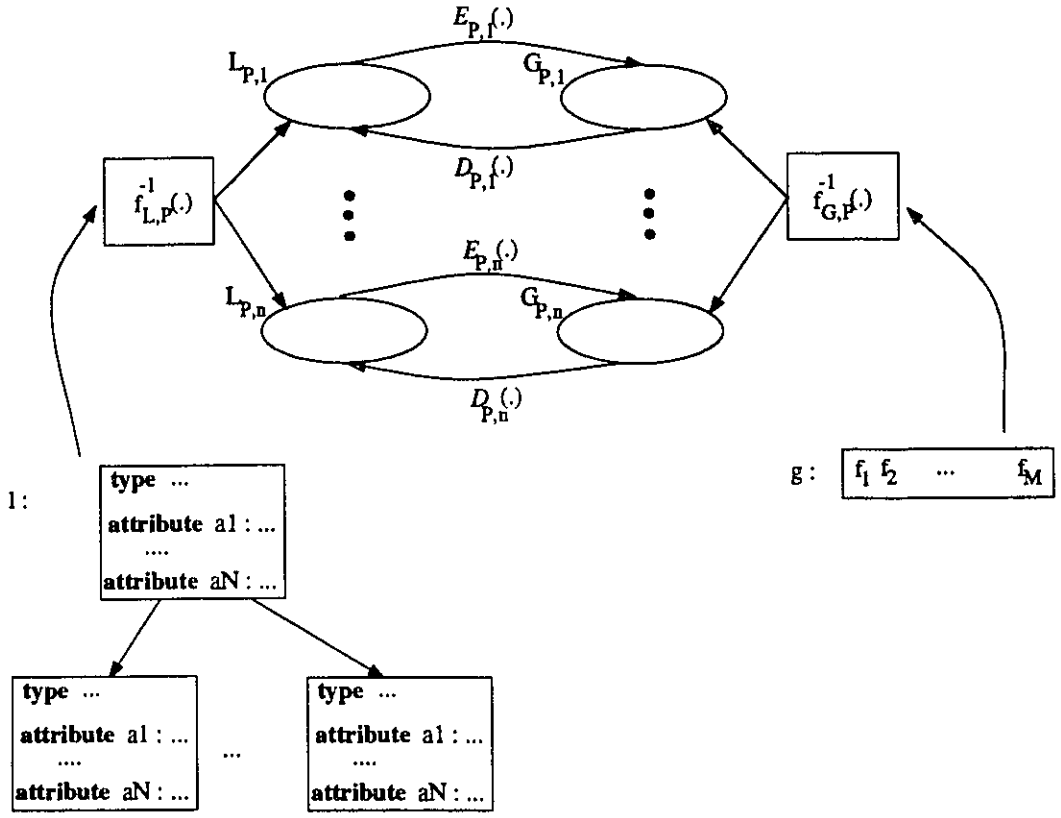
Similar to definition 2.3.1, definition 2.3.2 establishes the correctness criteria for encoding and decoding of PDU values of protocol P as :

1. $f^{-1}_{L,P}(l) = f^{-1}_{G,P}(\mathcal{E}_P(l))$
2. $f^{-1}_{G,P}(g) = f^{-1}_{L,P}(\mathcal{D}_P(g))$

where l and g are respectively of L_P and G_P.

Fig. 2.1 explains the encoding and decoding transformations according to definition 2.3.2, where n=|Tp|. According to the figure, nodes of local value l have N attributes and global value g consists of M functional units.

Figure 2.1 Encoding and Decoding Transformations According to Definition 2.3.2



According to definition 2.3.2.a, the first step of encoding transformation is to determine the type of the local value being encoded. Once the type has been identified then the encoding of the local value node(s) corresponding to the PDU type nodes is done. Since some components are defined to be optional, with default values, or to be chosen from a number of possible built-in types, a matching operation between nodes of the local value and the type becomes necessary. During the matching, functional units for nodes of local value are obtained. The result of such a matching operation is the intermediate value. The attributes of intermediate value nodes are the functional units of corresponding nodes of the local value. The last step of encoding is the serialization of functional units to generate the global value. If the serialization step is not independent of the PDU type, then intermediate value nodes have attributes regarding the rules about the serialization

step. Based on this partition, the encoding transformation can be rewritten as :

$$E_P(l) = \mathcal{E}_{P,f_{T,P}(f_{L,P}^{-1}(l))}(l) = \mathcal{A}_P(\mathcal{M}_{P,e}(l))$$

where $\mathcal{M}_{P,e}()$ is the transformation phase for the identification of PDU type and type-value matching, whereas, $\mathcal{A}_P()$ is the transformation phase for the serialization of functional units in intermediate value nodes, which we will shortly define as assembling.

Unlike the encoding transformation, decoding transformation may not always be separated into phases. The decoding transformation can be fully separated into two phases provided that functional units in a given global value can be identified independent of their related PDU type component. Based on this constraint, the decoding transformation can be obtained as the inverse of encoding :

$$D_P(g) = E_P^{-1}(g) = [\mathcal{A}_P(\mathcal{M}_{P,e}(g))]^{-1} = \mathcal{M}_{P,d}(\mathcal{P}_P(g))$$

where $\mathcal{M}_{P,d}()$ is the phase for the identification of type and type-value matching, whereas, $\mathcal{P}_P()$ is the phase for the separation of the global value into functional units of component instances, which we will shortly define as parsing. In the following section, parsing and assembling transformations are explained in detail.

2.3.2 Parsing and Assembling

The constraint given above to separate the decoding transformation into phases may be too rigid for some protocols. In this case, a partial parsing can be done when not all but some of the functional units are identifed without knowing their related PDU type component. In this case, the output of parsing transformation becomes a *hybrid form* of the PDU value instead of the intermediate form. The hybrid form is the two-tuple consisting of the global value and the partial intermediate value. In order to define the hybrid value, we first have to define what the partial intermediate value means in terms of *pre-order* traversal of trees.

Definition 2.3.3 : Let i be an intermediate value whose root is in_1 and having s ordered subtrees, $I_1, \ldots, I_s$. The pre-order traversal of i, i.e. $pre(i)$ is a list consisting of the node in_1, followed by nodes of I_1 in pre-order, followed by nodes of I_2 in pre-order, ..., followed by nodes of I_s in pre-order. Then $\widehat{i}$ is said to be related to i according to the relation **par** ($\widehat{i}$ **par** i) if $pre\left(\widehat{i}\right)$ is a prefix of $pre(i)$.

Based on the definition of relation **par**, we can define the hybrid form of a PDU value v.

Definition 2.3.4 : Let v be a PDU value, and i be a hybrid form of the value v. Then h is defined as a two-tuple, $h =< f_{G,P}(v), \widehat{i} >$ such that $\widehat{i}$ **par** $f_{I,P}(v)$.

According to definition 2.3.3, for an intermediate value i, there are a number of partial trees which may be related to i according to the relation **par**. However, the application of the inverse of encoding rules yields only one of them for each global value. Therefore, similar to definitions 2.2.1–2.2.3, we can define one-to-one onto functions mapping abstract values into their hybrid forms for each PDU type as well as any protocol.

Definition 2.3.5 : a) $f_{H,P,i}$ for PDU type $t_{P,i}$ of protocol P is a one-to-one onto function such that, $f_{H,P,i} : V_{P,i} \rightarrow H_{P,i}$ where $V_{P,i}$, and $H_{P,i}$ are value set, and hybrid value set of type $t_{P,i}$, respectively.

b) $f_{H,P}$ for protocol P is a one-to-one onto function $f_{H,P} : V_P \rightarrow H_P$ such that $f_{H,P}(v) = f_{H,P,f_{T,P}(v)}(v)$ where V_P, and H_P are value set, and hybrid value set of protocol P, respectively.

In the following definition, we define the assembling transformation in terms of global and intermediate forms. On the other hand, the parsing transformation is defined in terms

of global and hybrid forms. The parsing transformation becomes the dual of assembling only when the constraint about the identification of all functional units, independent of their PDU type components, is satisfied. This form of parsing is later described as a special case.

Definition 2.3.6 : a) *Assembling* for protocol P based on a set of assembling rules $[R_P]^a$ is a one–to–one onto function $\mathcal{A}_P : I_P \rightarrow G_P$ such that $\mathcal{A}_P(i) = g \Leftrightarrow f^{-1}_{I,P}(i) = f^{-1}_{G,P}(g)$,

b) *Parsing* for protocol P based on a set of assembling rules $[R_P]^a$ is a one–to–one onto function $\mathcal{P}_P : G_P \rightarrow H_P$ such that $\mathcal{P}_P(g) = h \Leftrightarrow f^{-1}_{G,P}(g) = f^{-1}_{H,P}(h)$,
where I_P, H_P, and G_P are intermediate value set, hybrid value set, and global value set of protocol P, respectively.

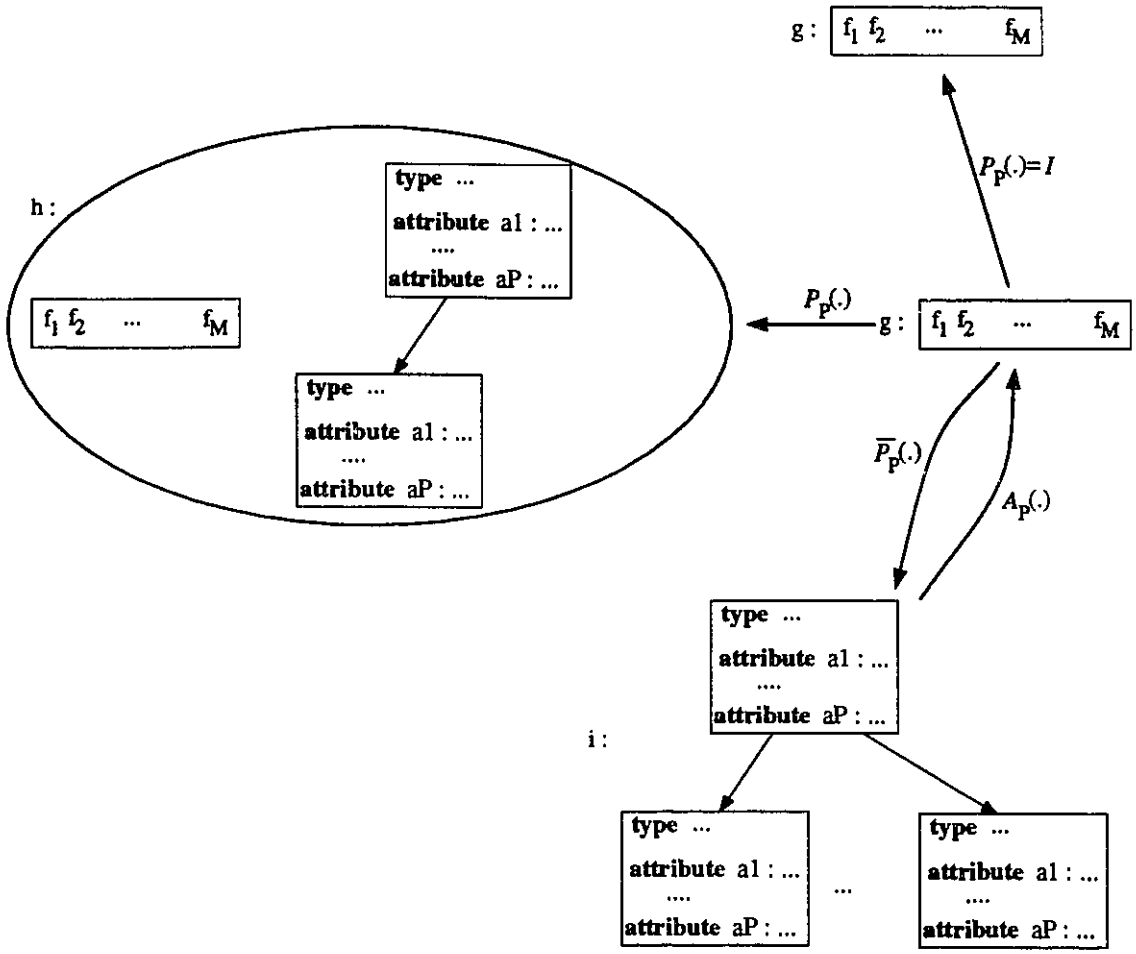
Fig. 2.2 shows the assembling transformation and three cases of parsing transformation in terms of intermediate, global and hybrid forms of PDU values. According to the figure, the nodes of the intermediate value i and the hybrid value l have P attributes. When the constraint about identifying the functional units independent of their related components of the PDU type is satisfied, the parsing transformation, i.e. $\overline{\mathcal{P}}_P$ results in an intermediate value. When this constraint is partially satisfied, the parsing transformation results in a hybrid value. When functional units for no local value node can be identified without knowing its related PDU type component, no parsing is possible.

The parsing transformation has two degenerate cases according to the composition of the hybrid form :

a) *Complete* parsing when there exists a rule set $[R_P]^a$ such that
$\forall h \in H_P \bullet \left(f_{H,P}(v) = h \Rightarrow h =< f_{G,P}(v), f_{I,P}(v) >\right)$ and is denoted by $\overline{\mathcal{P}}_P$.

b) *No* parsing when there exists a rule set $[R_P]^a$ such that

Figure 2.2 Parsing and Assembling Transformations According to Definition 2.3.6



$\forall h \in H_P \bullet \left(f_{H,P}(v) = h \Rightarrow h =< f_{G,P}(v), \epsilon >\right)$ where ϵ denotes the empty tree with no nodes. In this case the parsing transformation becomes equivalent to the identity function, i.e. $\mathcal{P}_P(.) = \mathcal{I}$.

Now having introduced the parsing and assembling transformations, we can explain the type-value matching transformations for encoding and decoding.

2.3.3 Type-Value Matching

In the following definition, the type-value matching transformation for encoding is defined in terms of local and intermediate forms. On the other hand, the type-value matching for decoding transformation is defined in terms of local and hybrid forms. The

type-value matching for decoding becomes the dual of that for encoding only when the parsing transformation is complete.

Definition 2.3.7 : a) *Type-value matching for encoding* for protocol P based on a set of type-value matching rules $[R_P]^m$ is a one–to–one onto function $\mathcal{M}_{P,e} : L_P \rightarrow I_P$ such that $\mathcal{M}_{P,e}(l) = i \Leftrightarrow f^{-1}_{L,P}(l) = f^{-1}_{I,P}(i)$,

b) *Type-value matching for decoding* for protocol P based on a set of type-value matching rules $[R_P]^a$ is a one–to–one onto function $\mathcal{M}_{P,d} : H_P \rightarrow L_P$ such that $\mathcal{M}_{P,d}(h) = l \Leftrightarrow f^{-1}_{H,P}(h) = f^{-1}_{L,P}(l)$, where L_P, and H_P are local value set, and hybrid value set of protocol P, respectively.

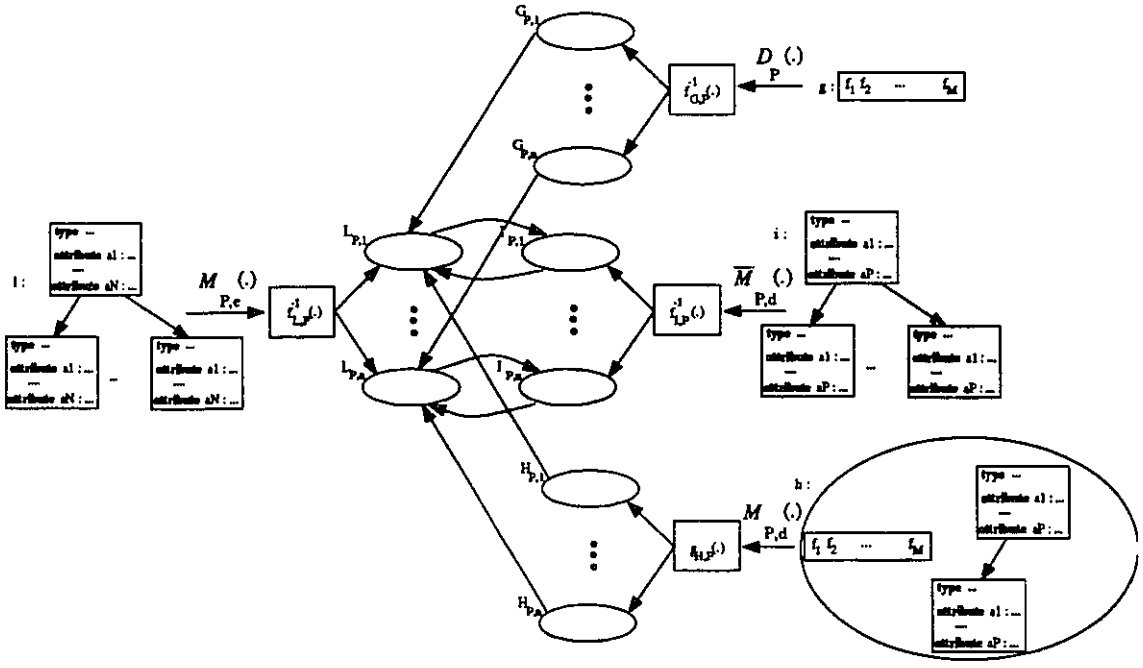
Fig. 2.3 shows the type-value matching transformation for encoding and three cases of the type-value matching transformation for decoding in terms of local, intermediate, hybrid, and global forms. When the parsing transformation is complete, the input of type-value matching for decoding is the intermediate value. When the parsing transformation is partial, the type-value matching transformation uses the hybrid value as its input. As the degenerate case, when no parsing takes place, the whole decoding is performed in just one phase.

Similar to parsing, type-value matching for decoding has two degenerate cases according to the composition of the hybrid form :

a) *Over* type–value matching for decoding when there exists a rule set $[R_P]^m$ such that $\forall h \in H_P \bullet \left(f_{H,P}(v) = h \Rightarrow h =< f_{G,P}(v), \epsilon >\right)$ where ϵ denotes the empty tree with no nodes. In this case, type-value matching for decoding becomes equivalent to D_P.

b) *Complete* type–value matching for decoding when there exists a rule set $[R_P]^m$ such that $\forall h \in H_P \bullet \left(f_{H,P}(v) = h \Rightarrow h =< f_{G,P}(v), f_{I,P}(v) >\right)$, and is denoted by $\overline{\mathcal{M}}_{P,d}$.

Figure 2.3 Type-Value Matching for Encoding and Transformations According to Definition 2.3.7



Based on the definitions for different cases of individual transformations for phases of encoding and decoding, we can conclude that the encoding transformation for any protocol P is always divisible into two phase transformations which are performed successively. On the other hand, the decoding transformation can be divided into two phases only if functional units for local value nodes can be identified without knowing their related PDU type nodes. The following definition gives the necessary conditions for decoding to become *(fully) divisible.*

Definition 2.3.8 : Decoding for protocol P based on a set of encoding rules R_P is *divisible* iff $\exists$ ($[R_P]^a$,$[R_P]^m$), a pair of rules sets, such that $\mathcal{P}_P(.) \neq \mathcal{I}$ and $\forall g \in G_P \bullet\ D_P(g) = \mathcal{M}_{P,d}(\mathcal{P}_P(g))$. If decoding based on R_P is divisible and $\forall g \in G_P \bullet\ D_P(g) = \overline{\mathcal{M}}_{P,d}(\overline{\mathcal{P}}_P(g))$, then decoding for protocol P is said to be *fully divisible.*

2.3.4 Further Refinements for Types and Local Values

As explained in the first chapter, PDUs themselves are abstractions used to convey the intended semantics of communication protocols during the dialogue. Therefore, a protocol implementation may require simpler data structures for protocol core functions than the types and values introduced in the previous sections. These functions may also require attributes other than the ones used in types and values. Such requirements necessitate another level of transformation between types and local forms of PDU values and their *refined* forms.

In this thesis, we do not include such transformations in the scope of encoding and decoding. The main reason for this approach is that the nature of processing for refinements and for encoding and decoding are quite different. While encoding and decoding are related with an external representation of PDU value, i.e. global value, refinements are done to perform conversions between data structures of different styles.

Now, having defined encoding and decoding transformations in terms of their phases and the conditions for divisibility of decoding transformation, we can introduce the algorithms for the phases in detail.

2.4 Decoding Algorithms

As explained in the previous section, each individual transformation for phases of decoding has two degenerate cases. When parsing transformation is equivalent to the identity function, the type-value matching for decoding transformation trivially becomes the entire decoding. However, instead of analyzing this degenerate case first, we try to introduce decoding as composed of two successive transformations. Therefore, in subsections 2.4.1 and 2.4.2, complete parsing and complete type-value matching for

decoding transformations are explained, respectively. In subsection 2.4.3, other cases of decoding, when the parsing transformation is not complete, are discussed.

2.4.1 Parsing

Parsing has been a topic of major interest in formal language theory. This is mainly because of its use in different areas such as compilers, pattern recognition, and natural language processing.

In [Aho 72], parsing is discussed in detail as one of the parts of the compiling process. According to [Aho 72], lexical analysis, symbol table operations, and parsing are initial steps of compiling. At the lexical analysis step, a given string of symbols is converted into syntactic entites, called *tokens*. The symbol table operations step deals with storing information about generated tokens. The parsing step uses a string of tokens which is the output of the lexical analyzer to determine whether the string obeys structural conventions of the syntactic definition of the language. The output obtained from the parser is a tree which represents the syntactic structure of the source program.

The functionality of PDU decoding is similar to the functionality of the lexical analysis and the parsing steps of compilers. In our approach, the parsing transformation for decoding combines lexical analysis and parsing of functional units according to the syntactic structure of the canonical representation. In the type-value matching transformation, intermediate values are syntactically analyzed according to the structures of related PDU types.

According to definition 2.3.6.b and the condition previously defined for parsing to be a complete transformation, we can define complete parsing transformation in terms of global and intermediate values.

Definition 2.4.1 : *Complete parsing* for protocol P based on a set of assembling rules $[R_P]^a$ is a one–to–one onto function $\overline{\mathcal{P}}_P : G_P \rightarrow I_P$ such that $\overline{\mathcal{P}}_P(g) = i \Leftrightarrow f^{-1}_{G,P}(g) = f^{-1}_{I,P}(i)$ where I_P, and G_P are intermediate value set, and global value set of protocol P, respectively.

As explained in the previous section, a global value is a string of functional units for nodes of the local value of that PDU. On the other hand, an intermediate value is represented with a typed, attributed tree whose nodes include each three functional units, i.e. identifier, length, and contents as their attributes. Therefore, the parsing transformation becomes the identification of functional units for each local value node in global value, and in turn the derivation of typed, attributed tree for the value being decoded.

Assume $\overline{\mathcal{P}}_P(g) = i$ where g is the string of functional units, i.e. $g = f_1 \ldots f_M$ and i has the root in_1 and s ordered subtrees $I_1, \ldots, I_s$ where every node in_j of i has attributes $in_j.id, in_j.lt, in_j.ct$. Then parsing transformation can be rewritten as a collection of transformations on functional units, $\overline{\mathcal{P}}_P(g) = \mathbf{P}_P(f_1)\,\mathbf{P}_P(f_2)\ldots\mathbf{P}_P(f_M)$ and the relations among individual transformations depend on the encoding rules of P.

The identification of functional units in a global value depend upon only the previously identified functional units when the parsing transformation is complete. The precedence relations between the transformations of functional units for a local value node and those between the transformations of functional units of different local value nodes are based on the encoding rules. Therefore, it is rather difficult to generalize these relationships. However, we can still make certain generalizations about the nature of these precedence relations.

Intermediate and local values, as well as PDU types, are represented by ordered

general trees based on the pointers *leftmost child* and *right sibling* [Aho 83]. Therefore, precedence relations between functional units for local value nodes must be such that the tree representing the intermediate value can be constructed from the global value.

Definition 2.4.2 : Let $g = f_1 \ldots f_M$ be a global value and functional units of g, $f_i = a_{i,1} \ldots a_{i,|f_i|}$, $f_j = a_{j,1} \ldots a_{j,|f_j|}$, be strings of atomic units. Then the parsing transformation of f_i is said to precede the parsing transformation of f_j, if $a_{j,1}$, and $func(f_j)$ can only be known after $\mathbf{P}_P(f_i)$. Here $func(f_j)$ specifies whether f_j is an identifier, length or contents unit. The preceding relation between transformations is denoted by $\prec$, i.e. $\mathbf{P}_P(f_j) \prec \mathbf{P}_P(f_i)$.

The precedence relations between functional units of a local value node change from protocol to protocol. However, there are certain characteristics common to all protocols:
a) For each local value node, there must be a non-empty functional unit set which includes units whose starting locations are defined by transformations of previously processed functional units for the other local value nodes. When all the functional units are of fixed size, this initial functional unit set includes all the functional units for the node.
b) Since definite length units are used to specify the locations of contents units of variable size, the transformation for a variable size contents unit of a local value node is always preceded by the transformation of the length unit of the local value.
c) Transformation of the contents unit for a local value node cannot precede those of an identifier or of length units of the node.

Let in_i be a node of the intermediate value tree i having children ordered from left to right, $in_{i+1}, \ldots, in_{i+nc(in_i)}$, where $nc(in_i)$ is the *number of children* of in_i. Let S_{i+j} be the initial functional unit set of the node in_{i+j} such that S_{i+j} includes the functional units whose transformations are preceded by transformations of functional units of other

nodes. Then, the precedence relations between functional units of different value nodes can be defined as one of the following two cases :

a) The transformations of functional units in S_{i+j} are preceded by the transformation of a functional unit of in_i, i.e. $\forall f \in S_{i+j} \bullet \left(\exists f' \; for \; in_i \bullet \mathbf{P}_P(f) \prec \mathbf{P}_P\left(f'\right)\right)$.

b) The transformations of functional units in S_{i+j} are preceded by the transformation of a functional unit of a sibling of in_{i+j}, e.g. n_{i+k} such that $k < j$,
i.e. $\forall f \in S_{i+j} \bullet \left(\exists f' \; for \; n_{i+k} \bullet \left(k < j \wedge \mathbf{P}_P(f) \prec \mathbf{P}_P\left(f'\right)\right)\right)$.

As explained before, in order to be able to parse a given global value, there needs to be an initial set of functional units whose functionality is known along with the intermediate value node to be related. Assume that S_0 is the initial set and the functions $\mathcal{N}_1$, and $\mathcal{F}_1$ define the node and functionality attribute of the unit in the intermediate value tree respectively. Then each functional unit transformation becomes the identification of the node and functionality attribute as well as a computation of a set of functional units whose node and functionality attribute can be computed at the next iteration step. In other words, such a set includes functional units whose transformations are preceded by the transformation of the current functional unit. Now, we can give the definition for transformations of functional units.

Definition 2.4.3 : Let g be a global value and f_i, f_j be functional units of g. The parsing transformation for the functional unit f_j is defined as

$$\mathbf{P}_P(f_j)=\begin{cases} <\mathcal{N}_k(f_j).\mathcal{F}_k(f_j),S_{k,j}> & f_j \in S_{k-1} \wedge \exists f_i \in S_{k,j} \bullet \mathbf{P}_P(f_i) \prec \mathbf{P}_P(f_j) \\ <\mathcal{N}_k(f_j).\mathcal{F}_k(f_j),\emptyset> & f_j \in S_{k-1} \wedge \forall f_i \; of \; g \bullet \neg(\mathbf{P}_P(f_i) \prec \mathbf{P}_P(f_j)) \end{cases}$$

where $\mathcal{N}_k(f_j)$, $\mathcal{F}_k(f_j)$ show the functions defining the node and functionality attribute of the unit f_j in the intermediate value tree at the k[th] iteration step of parsing. S_{k-1} shows the set of functional units whose transformations are preceded by the transformations which take place at the (k-1)[th] iteration step and $S_{k-1} = \bigcup_{f_j \in S_{k-2}} S_{k-1,j}$.

Notice that in definition 2.4.3, $\mathbf{P}_P(f_j)$ is only defined when $f_j \in S_{k-1}$. This shows that the parsing of each functional unit takes place at a certain iteration step of execution.

In [Aho 72], a number of different parsing algorithms for different classes of context free grammars (CFGs) are introduced. The Cocke-Younger-Kasami algorithm [You 67] has a $O(n^3)$ time and $O(n^2)$ space complexity for an n-size token string. For restricted classes of grammars, namely the LL(k) grammars, the LR(k) grammars, and the precedence grammars, sequential parsing algorithms for O(n) complexity are also given.

Since the nature of the grammar which would produce global values according to the encoding rules changes from protocol to protocol, it's not possible to give a specific type of parsing algorithm for the PDU decoding problem in general. Instead, we give an algorithm based on the parsing transformation definition for functional units in definition 2.4.3. An inherent problem with parsing is the possibility of having a string of atomic units which is correct according to the assembling rules, although not corresponding to an abstract value. Therefore, algorithm 2.1 cannot determine if a given string of atomic units is a global value for a protocol P; but it can determine that a given string of atomic units is not in the global value set G_P, due to a violataion of the assembling rules.

The algorithm of parsing transformation for each functional unit can be written recursively. However, instead, we give the algorithm to compute the overall parsing transformation, which is a collection of the individual transformations on functional units. In algorithm 2.1, set S_0 is used as the seed set of the computation where functions to compute the intermediate value node and the functionality attribute for each unit in S_0 are known. Set D is used to store the atomic units of the processed functional units during the parsing. Initially D is empty. When a functional unit f_j is found to be including an atomic unit $a_{j,i}$ which is previously processed as an atomic unit of another functional value, it means an error in the global value g. In other words, g is not a global value

included in the global value set G_P of protocol P. Similarly when the algorithm reaches to an iteration step where the set of functional units to be transformed at the next iteration step is empty, the set D should include all the atomic units of g. Otherwise, g is not in the global value set G_P of protocol P.

Algorithm 2.1 :

Step1. $k = 1, D = \emptyset$.

Step2. $\forall f_j \in S_{k-1}$ **do**

{**If** $\exists a_{j,i}\ of\ f_j \bullet a_{j,i} \in D$ **then** $\{g \notin G_P$. Stop.$\}$

else

{Compute $\mathcal{N}_k(f_j)$ and $\mathcal{F}_k(f_j)$.

Compute $S_{j,k}$.

$S_k = S_k \bigcup S_{j,k}$. $D = D \cup \left(\bigcup_{a_{j,i}\ of\ f_j} a_{j,i} \right)\}\}$

Step3. **If** $S_k \neq \emptyset$ **then** $\{k = k + 1$. Goto Step 2.$\}$

else

{**If** $D \neq \bigcup_{a_i\ of\ g} a_i$ **then** $\{g \notin G_P.\}$ Stop.$\}$

Since the precedence relations depend on the encoding rules of a protocol, the number of different threads during the parsing of a global value g changes from protocol to protocol. A degenerate case is when for a given global value $g = f_1 \ldots f_M$, $\forall f_i\ 1 \leq i \leq M \bullet f_i \in S_0$. In this case the entire parsing takes a single iteration step. Another degenerate case is when $\forall S_k\ k < M \bullet |S_k| = 1 \wedge |S_M| = 0$. Obviously, in this case the transformation of only one functional unit is performed at each iteration step and parsing takes M iteration steps.

Another interesting case is observed when all the length units use definite length format. In this case, when the transformation for a length unit of a node with a constructed

type is performed, the functional units for two other nodes become identifiable : the leftmost child and the right sibling of the node. The entire parsing transformation becomes equivalent to the multi-threaded pre-order traversal of the resulting intermediate value tree.

2.4.2 Type Value Matching for Decoding

Algorithm 2.1 yields a collection of atomic units of a given string of atomic units. This collection is an intermediate value if the given string is a global value which is syntactically correct according to the rules of canonical representation. However, since the same rules can be used to obtain canonical representations for values of different PDUs, the intermediate value should also be subjected to a structural analysis according to the rules of PDU types. On the other hand, if the given string is not a global value, the collection of atomic units obtained by algorithm 2.1 is not an intermediate value. This is also determined by the structural analysis. This second step is named as type-value matching for decoding, since at this step, nodes of a given intermediate value tree are compared with those of the type tree to obtain the corresponding local value.

Tree matching is a widely studied special problem which exists in such diverse areas; in programming theory [Hof 82], and molecular biology [Zha 89]. In [Hof 82], the tree pattern matching problem is given as locating the subtrees of a given *subject* tree that are isomorphic to some *pattern* tree possibly with variables standing for arbitrary subtrees. Bottom-up and top-down matching algorithms based on extensions of a fast string matching algorithm [Knu 77], are also given in [Hof 82]. The $O(mn)$ time bound of the naive algorithm for a pattern tree of size n and a subject tree of size m has been difficult to improve for the general case of this problem. Recently, an $O\left(nm^{0.75}polylog\left(m\right)\right)$ [Kos 89] and an $O\left(n\sqrt{m}polylog\left(m\right)\right)$ [Dub 90] have been given for this problem. However, all these algorithms are multistep since they require preprocessing of trees, such as converting n-ary trees to their equivalent binary trees or

calculating representative trees to be used in comparisons. Such methods increase the space complexity and are not very effective when tree sizes are relatively small. Another factor for us not to have chosen an existing algorithm for this step is that in our case there is always one and only one match between type and intermediate value trees if the given atomic unit collection is in the intermediate value set I_P of protocol P. Therefore, for our type-value matching problem we developed an algorithm which does not require any preprocessing of given trees.

According to definition 2.3.7.b and the condition previously defined for parsing to be a complete transformation, we can define complete type-value matching for decoding transformation in terms of intermediate and local values.

Definition 2.4.4 : *Complete type-value matching for decoding* for protocol P based on a set of type-value matching rules $[R_P]^m$ is a one–to–one onto function $\overline{\mathcal{M}}_{P,d} : I_P \rightarrow L_P$ such that $\overline{\mathcal{M}}_{P,d}(i) = l \Leftrightarrow f^{-1}_{I,P}(i) = f^{-1}_{L,P}(l)$ where I_P, and L_P are intermediate value set, and local value set of protocol P. respectively.

Now, before trying to explain the algorithm for type-value matching for decoding, let us formalize the tree matching problem for our purposes. It should be noted that, in type-value matching for encoding and decoding, all PDU types of a protocol are combined into a single tree. This tree has a root without attributes and its children are the root nodes of trees for PDU types. In the following definitions, T_P is used to denote this protocol type tree.

Definition 2.4.5 : A node in_i of tree i matches to node tn_j of T_P iff their identifier attributes are equivalent, i.e if there is a function $eq(.)$ such that $eq(in_i.id) = tn_j.id$ and for any child in_{i+k} of in_i, there is a child tn_{j+l} of it_j such that in_{i+k} matches it_{j+l}.

Based on definition 2.4.5, the type-value matching for decoding transformation can be rewritten as a collection of transformations on intermediate value nodes. Assume $i = in_1, \ldots, in_n$ where in_i shows an individual node of i, then $\overline{\mathcal{M}}_{P,d}(i) = \mathrm{M}_{\mathrm{P,d}}(in_1) \ldots \mathrm{M}_{\mathrm{P,d}}(in_n)$ and the relations among individual transformations follow the pre-order traversal of i.

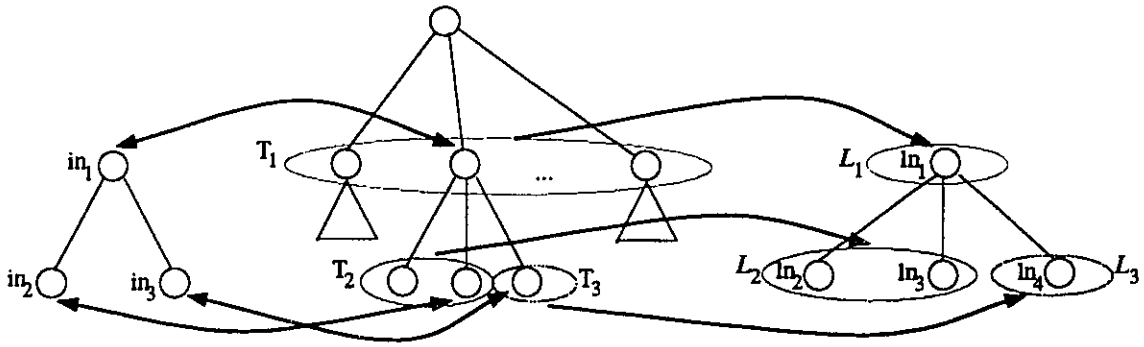
Our type-value matching for decoding algorithm is based on a top-down traversal of the intermediate value tree. Therefore, the algorithm starts with the root node in_1 of i. In order to find the matching type tree node for in_1, there needs to be a set T_1 including type tree nodes which are possible matches for in_1. Based on the explanation of T_P, T_1 consists of the root nodes of all PDU types of protocol P. However, in the algorithms of type-value matching for encoding and decoding, in order not to increase the space complexity of the algorithm, instead of T_1, a *handle* for T_1 is used. Such a handle includes a *seed* node and the method to generate all the other nodes in T_1 by traversing T_P. Nevertheless, in order to simplify the definitions, we use the set itself in the following discussion. Since there is one and only one PDU type whose root node has an equivalent identifier to the identifier of in_1, this node tn_i is found as the *potential match* for in_1. We use the word potential since at this point of execution it is not known if all the children of in_1 have matching nodes among the children of tn_i. For these potentially matching nodes the root node ln_1 of local value l is generated.

Once a potential match type node is found for in_1, the set T_2 of possible type nodes for the leftmost child in_2 of in_1 is computed. Unlike the root node in_1, T_2 for in_2 may

consist of a single type tree node if the instance of this node should always be present in the local value tree as the leftmost child of the root node. If the potential match for in_1, tn_i is of the set type, then its children may appear in the intermediate values in any order. Therefore, in this case T_2 includes all children of tn_i. If tn_i has a leftmost child which is of default or optional type, then T_2 includes all the children of tn_i, ordered from left to right until reaching the first child which is not optional or default. If all the children are optional or default, then all of them are included in this set. In this case even if an optional or default type node is not a potential match for in_2, a local value node is still generated to specify the exclusion of the instance. Once the potential match for in_2 is found, then the sets of possible type nodes for right sibling and leftmost child of in_2 can be computed.

In order to better illustrate type-value matching for decoding, let us examine a simple example. In Fig. 2.4, an intermediate value tree i with 3 nodes is given to be matched with the type tree T_P of protocol P. According to the figure, the root node in_1 of i, is to be matched with the children of the root node of T_P. When the second node of the root is found to be a potential match for in_1, the root node ln_1 of local value l is generated. Assume that the instance of leftmost child of the potential match for in_1 is declared as optional. In this case, both this node and its right sibling which is not an optional or default node are included in T_2. If the optional node is not a potential match for in_2, a node of l is still created to specify that its instance is not present. In this case, the potential match for in_2 is the right sibling of the optional node. Similary, the node created for this possible match pair is the right sibling of ln_2. At the next iteration step, the potential match for in_3 is found in T_3. In the figure, $\mathcal{L}_i$ denotes the set of local value nodes generated while the possible match for in_i is searched for in the set T_i.

Figure 2.4 An Example for Type-Value Matching for Decoding



Now, before trying to define the type-value matching for decoding transformation of each intermediate value node, we have to define the function $\mathcal{L}(in_j, T_j)$ which generates the local value nodes for (in_j, T_j) pairs. When there is a potential match for in_j in the set T_j, the function $\mathcal{IT}(in_j, T_j)$ defines the intermediate value node and set for possible match pairs to be processed at the next iteration step of the algorithm.

In the following definition, the set of possible matches for an intermediate value node always consist of some or all of the children of a single type tree node. The set of possible matches may be either of type Choice or Set or Sequence depending on the type of the parent type tree node. When the parent node is of type Choice, there is one and only one potential match in the set for the intermediate value node. When the parent node is of type Set, again there is one and only one potential match for the intermediate value node. When the parent node is of type Sequence, the output of the type-value matching for decoding is somewhat different. In this case, the children of the parent node is evaluated from left to right as a potential match for the intermediate value node. This evaluation is done until either a potential match is found or a child which is neither Optional nor Default (O/D) is evaluated and it is not found to be a potential match for the intermediate value node.

Definition 2.4.6 : Let in_j be a node of the intermediate value i and T_j be the set of possible matches for in_j in the type tree T_P of protocol P. Then the function which generates the local value nodes can be defined as:

$$\mathcal{L}(in_j,T_j)=\begin{cases} ln_k & \exists tn_i \in T_j \bullet eq(in_j)=tn_i \wedge T_j \text{ is either } CHOICE \text{ or } SET \text{ type} \wedge in_j \neq \epsilon \\ \{ln_k,\ldots,ln_{k+m}\} & (T_j=\{tn_i,\ldots,tn_{i+n}\} \wedge \forall x<m \leq n \bullet (eq(in_j) \neq tn_{i+x} \wedge tn_{i+x} \text{ is } O/D) \wedge \\ & eq(in_j)=tn_{i+m}) \vee (T_j \text{is } SET \text{ type} \wedge in_j=\epsilon \wedge |T_j|=m+1 \wedge (\forall tn_i \in T_j \bullet tn_i \text{ is } O/D)) \\ error & \forall tn_i \in T_j \bullet eq(in_j) \neq tn_i \end{cases}$$

Now based on definition 2.4.6, we can define the function which generates the pairs of an intermediate value node and its possible match set to be processed at the next iteration step. In definition 2.4.7, following predicates for (in_j, T_j) pairs are used to shorten the definition.

$$pre_1(in_j,T_j)=(\mathcal{L}(in_j,T_j) \neq error)$$

$$pre_2(in_j,T_j)=(\exists tn_k \in T_j \bullet eq(in_j)=tn_k) \wedge T_j \text{ is } CHOICE \text{ type}$$

$$pre_3(in_j,T_j)=(\exists tn_k \in T_j \bullet eq(in_j)=tn_k) \wedge T_j \text{ is } SET \text{ type}$$

$$pre_4(in_j,T_j)=(T_j=\{tn_i,\ldots,tn_{i+n}\} \wedge \forall x<m \leq n \bullet (eq(in_j) \neq tn_{i+x}) \wedge eq(in_j)=tn_{i+m})$$

Similarly, we define predicates pre_5 and pre_6 for intermediate and type tree nodes to specify whether they have children and right sibling using predicates . The remaining three predicates specify whether a type tree node is of Choice, Set, or Sequence type.

$$pre_5(*)=(* \text{ has children}) \qquad pre_7(tn_k)=(tn_k \text{ is } CHOICE \text{ type})$$

$$pre_6(*)=(* \text{ has right sibling}) \qquad pre_8(tn_k)=(tn_k \text{ is } SET \text{ type})$$

$$pre_9(tn_k)=(tn_k \text{ is } SEQUENCE \text{ type})$$

Based on these predicates, we may define the predicate which identifies the erronous case during the generation of pairs of an intermediate value node and its possible match set.

$$p_{error}=(\neg pre_1(in_j,T_j))\vee(pre_5(in_j)\wedge(\neg pre_5(tn_k)))\vee$$

$$((\neg pre_5(in_j)\wedge(\exists tn_m \bullet tn_m \textit{ child of } tn_k\wedge\neg tn_m \textit{ is } O/D)))\vee$$

$$(pre_6(in_j)\wedge(pre_3(in_j,T_j)\wedge T_j-\{tn_k\}=\emptyset))\vee(pre_6(in_j)\wedge(pre_4(in_j,T_j)\wedge(\neg pre_6(tn_k))))\vee$$

$$(\neg pre_6(in_j)\wedge(pre_3(in_j,T_j)\vee pre_4(in_j,T_j))\wedge(\exists tn_m \bullet tn_m\in(T_j-\{tn_k\})\wedge\neg tn_m \textit{ is } O/D))$$

Definition 2.4.7 : Let in_j be a node of the intermediate value i, T_j be the set of possible matches for in_j in the type tree T_P of protocol P, and tn_k be the potential match for in_j in T_j. Then the function which generates the pairs of an intermediate value node and its possible match set can be defined as :

$$IT(in_j,T_j)=\begin{cases} \{(in_{lc},T_{lc}),(in_{rs},T_{rs})\} & pre_5(in_j)\wedge pre_6(in_j)\wedge(\neg p_{error}) \\ \{(in_{lc},T_{lc}),(\epsilon,T_{rs})\} & pre_5(in_j)\wedge(\neg pre_6(in_j))\wedge(T_{rs}\neq\emptyset)\wedge(\neg p_{error}) \\ \{(in_{lc},T_{lc})\} & pre_5(in_j)\wedge(\neg pre_6(in_j))\wedge(T_{rs}=\emptyset)\wedge(\neg p_{error}) \\ \{(\epsilon,T_{lc}),(in_{rs},T_{rs})\} & (\neg pre_5(in_j))\wedge pre_6(in_j)\wedge(T_{lc}\neq\emptyset)\wedge(\neg p_{error}) \\ \{(in_{rs},T_{rs})\} & (\neg pre_5(in_j))\wedge pre_6(in_j)\wedge(T_{lc}=\emptyset)\wedge(\neg p_{error}) \\ \{(\epsilon,T_{lc}),(\epsilon,T_{rs})\} & (\neg pre_5(in_j))\wedge(\neg pre_6(in_j))\wedge(T_{lc}\neq\emptyset)\wedge(T_{rs}\neq\emptyset)\wedge(\neg p_{error}) \\ \{(\epsilon,T_{lc})\} & (\neg pre_5(in_j))\wedge(\neg pre_6(in_j))\wedge(T_{lc}\neq\emptyset)\wedge(T_{rs}=\emptyset)\wedge(\neg p_{error}) \\ \{(\epsilon,T_{rs})\} & (\neg pre_5(in_j))\wedge(\neg pre_6(in_j))\wedge(T_{lc}=\emptyset)\wedge(T_{rs}\neq\emptyset)\wedge(\neg p_{error}) \\ \emptyset & (\neg pre_5(in_j))\wedge(\neg pre_6(in_j))\wedge(T_{lc}=\emptyset)\wedge(T_{rs}=\emptyset)\wedge(\neg p_{error}) \end{cases}$$

where the possible match set for the leftmost child in_{lc} is

$$T_{lc}=\begin{cases} \bigcup_{i=1}^{nc(tn_k)}\{tn_{k+i}\} & pre_7(tn_k)\vee pre_8(tn_k) \\ \{tn_{k+1},\ldots,tn_{k+n}\} & (pre_9(tn_k)\wedge\forall x<n\bullet tn_{k+x}\ is\ O/D\wedge(\neg tn_{k+n}\ is\ O/D))\vee \\ & (pre_9(tn_k)\wedge\forall x\leq n\bullet tn_{k+x}\ is\ O/D\wedge nc(tn_k)=n) \end{cases}$$

and the possible match set for the right sibling in_{rs} is

$$T_{rs}=\begin{cases} T_j-\{tn_k\} & pre_3(in_j,T_j) \\ \{tn_{k+1},\ldots,tn_{k+n}\} & (pre_4(in_j,T_j)\wedge\forall x<n\bullet tn_{k+x}\ is\ O/D\wedge(\neg tn_{k+n}\ is\ O/D))\vee \\ & (pre_4(in_j,T_j)\forall x\leq n\bullet tn_{k+x}\ is\ O/D\wedge nc(tn_k)=n) \end{cases}$$

According to definition 2.4.7, when an intermediate value node does not have a right sibling or any children, there may still be remaining type tree nodes to be processed. In

order to handle these cases, the function $\mathcal{IT}()$ generates an empty intermediate value node with a set of remaining type nodes. Although there is no possible match for an empty intermediate value, the local value nodes for instances of optional and default type nodes need to be generated.

Based on the previous definitions, we can give the definition for type-value matching transformations for each intermediate value tree node.

Definition 2.4.8 : Let $i = in_1, \ldots, in_n$ be an intermediate value and in_j be a node of i. The type-value matching for decoding transformation for in_j is defined as

$$\mathbf{M}_{P,d}(in_j) = \begin{cases} <\mathcal{L}(in_j,T_j),\mathcal{IT}(in_j,T_j)> & \mathcal{L}(in_j,T_j) \neq error \wedge \mathcal{IT}(in_j,T_j) \neq error \\ error & otherwise \end{cases}$$

The algorithm for type-value matching for decoding is the collection of individual transformations as in the case of algorithm 2.1. In algorithm 2.2, set IT_0 is used as the seed set of the computation where $IT_0 = \{(in_1, T_1)\}$. At each iteration step of the algorithm, the type-value matching for decoding transformation for each intermediate value node of the pairs in set U is computed. When the algorithm reaches to an iteration step where U is empty, the algorithm terminates by deciding that i is in the intermediate value set I_P.

Algorithm 2.2 :

Step1. $U = IT_0, V = \emptyset$.

Step2. $\forall (in_j, T_j) \in U$ **do**

{Compute $\mathcal{L}(in_j, T_j)$.

If $\mathcal{L}(in_j, T_j) = error$ **then** {$i \notin I_P$. Stop.}

else

{Compute $\mathcal{IT}(in_j, T_j)$.

If $\mathcal{IT}(in_j, T_j) = error$ **then** {$i \notin I_P$. Stop.}

else {$V = V \bigcup \mathcal{IT}(in_j, T_j)$.}}}

Step3. $U = V, V = \emptyset$.

Step4. **If** $U \neq \emptyset$ **then** {Goto Step 2.} **else** {$i \in I_P$. Stop.}

The time complexity of algorithm 2.2 varies with the characteristics of type tree T_P of protocol P. In terms of the size of the intermediate value tree and possible match sets, the time complexity is $O(n\,|T_i|)$. If there is only one possible match for each intermediate value node, e.g. $|T_i| = 1$, then the time complexity of algorithm 2.2 becomes $O(n)$. The upper-bound for the average size of T_i is m when T_i includes all nodes of type tree T_P. In this case, the time complexity of algorithm 2.2 becomes $O(nm)$. However, in most cases the time complexity is in between these two extremes. In the next chapter, the complexity of distributed implementations of algorithms introduced in this chapter are discussed in greater detail.

2.4.3 Decoding with Partial Parsing

In the previous subsections, we discussed decoding in terms of complete phase transformations. However, as explained in section 2.3, the parsing transformation may be partial when only some of the nodes for an internal value can be generated from a

global value without knowing the PDU type. As a degenerate case, entire decoding is done in a single phase when no functional unit in a given global value can be identified without knowing its related type.

When the parsing transformation is partial, certain changes need to be made on algorithms 2.1 and 2.2. In this case, step 3 of algorithm 2.1 is updated such that when $S_k = \emptyset$, the algorithm stops whithout issuing a verdict whether or not $g \in G_P$. This is due to the fact that not all functional units in g can be identified and associated with an intermediate value node. Similarly step 2 of algorithm 2.2 is updated such that when a leftmost child or a right sibling for a node of $\widehat{i}$ of a given hybrid value $h =< g, \widehat{i} >$ is not present in $\widehat{i}$, its functional units are searched and identified in g to generate the node.

When no functional unit in a given global value g can be identified without knowing its related type node, algorithms 2.1 and 2.2 are combined into a single algorithm. In this case, the seed set of the computation includes the pairs of functional units and their possible type nodes.

2.5 Encoding Algorithms

Unlike decoding, phases of encoding are always complete transformations. Therefore, we analyze encoding in terms of its phase transformations, i.e. type-value matching for encoding and assembling in subsections 2.5.1 and 2.5.2, respectively.

2.5.1 Type Value Matching for Encoding

The duality which exists between encoding and decoding of PDUs, also exists between the respective phase transformations of encoding and decoding. The first phase tranformation of encoding, i.e. type-value matching for encoding is the dual of the second phase transformation of decoding, i.e. type-value matching for decoding.

According to definition 2.3.7.a, the type-value matching for encoding transformation is applied on local values in the local value set of a protocol to generate the intermediate values. Similar to type-value matching for decoding, the type tree Tp of a protocol P is used in the transformation to generate the intermediate values.

Similar to type-value matching for decoding, the type-value matching for encoding transformation can be written as a collection of transformations on local value nodes. Assume $l = ln_1, \ldots, ln_n$ where ln_i shows an individual node of l, then $\mathcal{M}_{P,e}(l) = \mathrm{M}_{\mathrm{P},e}(ln_1) \ldots \mathrm{M}_{\mathrm{P},e}(ln_n)$ and the relations among individual transformations follow the pre-order traversal of l.

Similar to algorithm 2.2, the algorithm for type-value matching for encoding is based on a top-down traversal of local value tree. Therefore, the algorithm starts with the root node ln_1 of l. T_1 includes the type tree nodes which are possible matches for ln_1. Similar to algorithm 2.2, once a potential match is found for a local value node ln_i and an intermediate value node is generated, at the next step the leftmost child (if it exists) and the right sibling (if it exists) of ln_i are processed to find their potential matches.

Unlike the type-value matching for decoding transformation, the type-value matching for encoding transformation for each local value node results in at most one intermediate value node. Since local values include nodes of *nil* value declared as optional or with default values, type-value matching for encoding does not always generate an intermediate value node for each local value node.

Another important difference between type-value matching for encoding and decoding is that while an intermediate value tree is generated starting from its root, a bottom-up traversal of nodes is also necessary. The bottom-up traversal is used to propagate the information about the functional unit attributes of intermediate value nodes to their

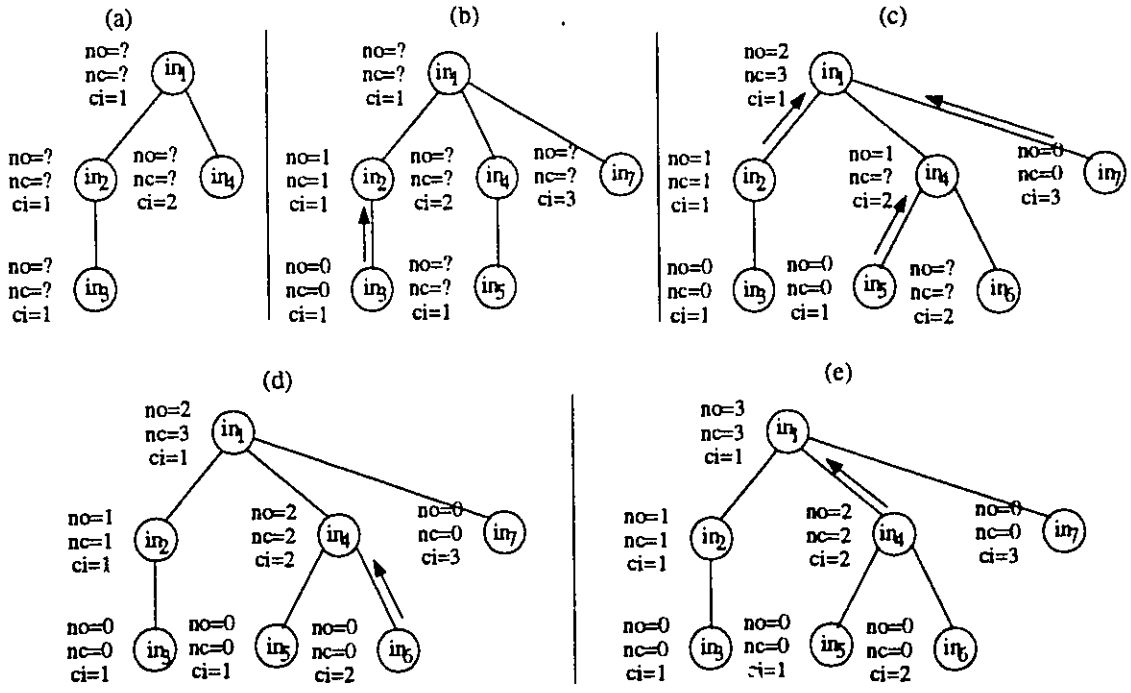
parents. Such information is usually the size of functional units, for when the length of a value of a constructed type is encoded in definite format, it should provide the size of the functional units of its components.

Type-value matching for encoding generates the intermediate value tree by using leftmost child and right sibling pointers as in the case of previous algorithms. However, bottom-up traversal of the intermediate value tree requires additional pointers and counters. Type-value matching for encoding generates a pointer for each intermediate value node to its parent. Therefore, when (ln_{j+1}, T_{j+1}) is generated from the processing of (ln_j, T_j), where ln_{j+1} is the leftmost child of ln_j, a pointer to the intermediate value node in_k which is generated for (ln_j, T_j) is also passed. The necessary counters of the computation are the total number of children counter, i.e. $nc(in_j)$ and the counter to register the number of children from which the propagating information is received, i.e. $in_j.no$. Since an intermediate value tree is generated top-down, when a node in_j is generated the total number of its children $nc(in_j)$ is not known. Therefore, this counter can only be set after the generation of its rightmost child which does not have any right sibling of its own. In order to count the children of a node ordered from left to right, the type-value matching for encoding algorithm uses a child identifier $in_j.ci$ associated with each node in_j while generating each intermediate value node. In order to propagate the information from children to their parents, an upwards procedure is used in the type-value matching for encoding algorithm. When information needed is carried from a child to its parent, e.g. in_j the counter $in_j.no$ is incremented. The propagation of information from all children is completed for a node when the counter $in_j.no$ becomes equal to the counter $nc(in_j)$. Then the information from in_j can be propagated to its parent. This procedure is repeated until the information from all its children is propagated back to the root node in_1 of i.

Fig. 2.5.(a)-(e) explains the upward propagation of information on an example

intermediate value tree i. Initially, in Fig. 2.5.(a), only four nodes of i are generated. In Fig. 2.5.(b), since $nc(in_3) = in_3.no = 0$ information about in_3 is propagated up to its parent in_2. In Fig. 2.5.(c) information from both in_2 and in_7 is propagated up to the root node in_1. Since in_7 is the rightmost sibling of in_1, $nc(in_1)$ is also set to 3 at this step. Upward propagation terminates when information from in_4 is propagated up to in_1 as shown in Fig. 2.5.(e).

Figure 2.5 An Example for Upward Propagation



Now, similar to definitions given for complete type-value matching for decoding transformation, we can give the definitions for functions used in the type-value matching for encoding transformation.

Definition 2.5.1 : Let ln_j be a node of the local value l and T_j be the set of possible matches for ln_j in the type tree T_P of protocol P. Then the function which generates the intermediate value nodes can be defined as :

$$\mathcal{I}(ln_j,T_j)=\begin{cases} in_k & (\exists tn_i \in T_j \bullet eq(ln_j)=tn_i \wedge T_j \textit{ is either } CHOICE \textit{ or } SET \textit{ type}) \vee \\ & (T_j=\{tn_i\} \wedge eq(ln_j)=tn_i \wedge (\neg ln_j \textit{ is } O/D)) \\ error & \forall tn_i \in T_j \bullet eq(ln_j) \neq tn_i \end{cases}$$

Now based on definition 2.5.1, we can define the function which generates the pairs of local value node and its possible match set to be processed at the next iteration step. In definition 2.5.2, the following predicates for (ln_j, T_j) pairs are used to shorten the definition.

$$pre_1(ln_j,T_j)=(\mathcal{I}(ln_j,T_j)\neq error)$$

$$pre_2(ln_j,T_j)=(\exists tn_k \in T_j \bullet eq(ln_j)=tn_k) \wedge T_j \textit{ is } CHOICE \textit{ type}$$

$$pre_3(ln_j,T_j)=(\exists tn_k \in T_j \bullet eq(ln_j)=tn_k) \wedge T_j \textit{ is } SET \textit{ type}$$

$$pre_4(ln_j,T_j)=(T_j=\{tn_k\} \bullet eq(ln_j)=tn_k)$$

The predicates $pre_5, \ldots, pre_9$ defined in section 2.4.2 are also used for type-value matching for encoding. Based on these predicates, we may define the predicate which identifies the erronous case during the generation of pairs of a local value node and its possible match set.

$$p_{error}=(\neg pre_1(ln_j,T_j)) \vee (pre_5(ln_j) \wedge (\neg pre_5(tn_k))) \vee$$

$$((\neg pre_5(ln_j) \wedge (\exists tn_m \bullet tn_m \textit{ child of } tn_k \wedge \neg tn_m \textit{ is } O/D))) \vee$$

$$(pre_6(ln_j) \wedge (pre_3(ln_j,T_j) \wedge T_j-\{tn_k\}=\emptyset)) \vee (pre_6(ln_j) \wedge (pre_4(ln_j,T_j) \wedge (\neg pre_6(tn_k)))) \vee$$

$$(\neg pre_6(ln_j) \wedge (pre_3(ln_j,T_j) \vee pre_4(ln_j,T_j)) \wedge (\exists tn_m \bullet tn_m \in (T_j-\{tn_k\})))$$

Definition 2.5.2 : Let ln_j be a node of the local value l, T_j be the set of possible matches for ln_j in the type tree T_P of protocol P, and tn_k be the potential match for ln_j in T_j. Then the function which generates the pairs of a local value node and its possible match set can be defined as :

$$\mathcal{LT}(ln_j,T_j)=\begin{cases}\{(ln_{lc},T_{lc}),(ln_{rs},T_{rs})\} & pre_5(in_j)\wedge pre_6(in_j)\wedge(\neg p_{error})\\ \{(ln_{lc},T_{lc})\} & pre_5(in_j)\wedge(\neg pre_6(in_j))\wedge(\neg p_{error})\\ \{(ln_{rs},T_{rs})\} & (\neg pre_5(in_j))\wedge pre_6(in_j)\wedge(\neg p_{error})\\ \emptyset & (\neg pre_5(in_j))\wedge(\neg pre_6(in_j))\wedge(\neg p_{error})\end{cases}$$

where the possible match set for the leftmost child ln_{lc} is

$$T_{lc}=\begin{cases}\bigcup_{i=1}^{nc(tn_k)}\{tn_{k+i}\} & pre_7(tn_k)\vee pre_8(tn_k)\\ \{tn_{k+1}\} & pre_9(tn_k)\end{cases}$$

and the possible match set for the right sibling ln_{rs} is

$$T_{rs}=\begin{cases}T_j-\{tn_k\} & pre_3(ln_j,T_j)\\ \{tn_{k+1}\} & pre_4(ln_j,T_j)\end{cases}$$

Based on the previous definitions, we can give the definition for type-value matching for encoding transformations for each local value tree node.

Definition 2.5.3 : Let $l = ln_1,\ldots,ln_n$ be a local value and ln_j be a node of l. The type-value matching for encoding transformation for ln_j is defined as

$$\mathbf{M}_{T,n}(ln_j)=\begin{cases}<\mathcal{I}(ln_j,T_j),\mathcal{LT}(ln_j,T_j)> & \mathcal{I}(ln_j,T_j)\neq error\wedge\mathcal{LT}(ln_j,T_j)\neq error\\ error & otherwise\end{cases}$$

In algorithm 2.3, the set LT_0 is used as the seed set of the computation where $LT_0 = \{(\{ln_1\},T_1)\}$. At each iteration step of the algorithm, the type-value matching for encoding transformation for each local value node of the pairs in set U is computed. Also, from each generated intermediate value node an upward propagation is started. Propagation stops when the root node in_1 is encountered. When the algorithm reaches

an iteration step where U is empty, the algorithm terminates by deciding that l is in the intermediate value set L_P.

Algorithm 2.3 :

Step1. $U = LT_0, V = \emptyset$.

Step2. $\forall (ln_j, T_j) \in U$ **do**

{Compute $\mathcal{I}(ln_j, T_j)$

If $\mathcal{I}(ln_j, T_j) = error$ **then** {$l \notin L_P$. Stop.}

else

{Compute $\mathcal{LT}(ln_j, T_j)$.

If $\mathcal{LT}(ln_j, T_j) = error$ **then** {$l \notin L_P$. Stop.}

else

{**If** $(\neg pre_5(ln_j)) \wedge (j \neq 1)$ **then** {Upward($\mathcal{I}(ln_j, T_j)$).}

$V = V \cup \mathcal{LT}(ln_j, T_j)$}}}

Step3. $U = V, V = \emptyset$.

Step4. **If** $U \neq \emptyset$ **then** {Goto Step 2.} **else** {$l \in L_P$. Stop.}

Upward(in_j) : {Update $parent(in_j).lt$.

$parent(in_j).no = parent(in_j).no + 1$.

If $\neg pre_6(in_j)$ **then** $\{nc(parent(in_j)) = in_j.ci.\}$

If $(parent(in_j).no = nc(parent(in_j))) \wedge (parent(in_j) \neq in_1)$ **then**

{Upward($parent(in_j)$).}

else {Return.}}

2.5.2 Assembling

The assembling transformation is the final phase of encoding. In this phase, functional units associated with each intermediate value node are serialized to obtain the global value. According to definition 2.3.6.a, the assembling transformation is applied on intermediate values in the intermediate value set of protocol P to generate global values.

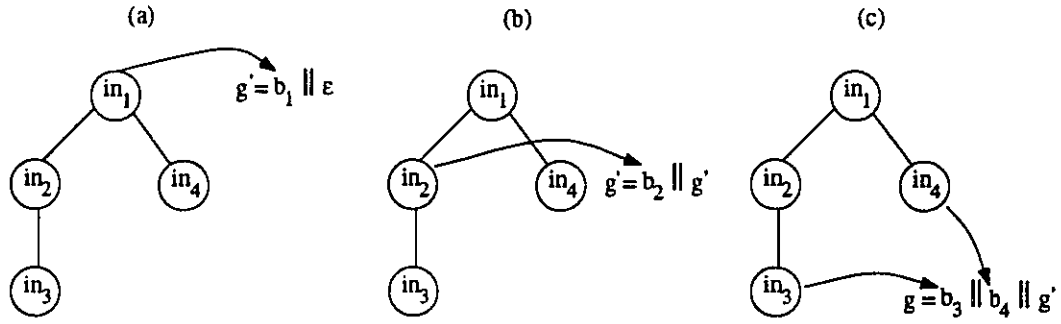
Unlike its dual, the assembling transformation does not check whether a given intermediate value i is in the intermediate value set I_P of a protocol P. Since the first phase of encoding, the type-value matching for encoding transformation, generates the intermediate value i for a given local value l only if l is in the local value set L_P, it is guaranteed that i is in I_P.

In order to generate the global value g for a given intermediate value i, a pre-order traversal of the intermediate value tree is performed. For each intermediate value node, the functional units to be included in the global value are written at addresses passed from parents to their leftmost children and nodes to their right sibling. In order to formalize the generation of global values, we can define a function which generates bit streams for a given intermediate value, address pair. The bit stream b_j includes the functional units of an intermediate value node in_j at locations specified by adr_j and the relative positions of the functional units. Based on this formalism, in order to obtain the global value g, a logical OR operation needs to be performed on the bit streams generated for different intermediate value nodes at each iteration step and the partial global value constructed up to that step. In order to better describe the generation of a global value g from an intermediate value i, let us examine Fig. 2.6.a-c. In Fig. 2.6.a, the first partial global value is constructed as the logical OR of an empty string ϵ and the bit string b_1 obtained for the root node in_1 of i. In Fig. 2.6.b, the same operation is shown for the leftmost child in_2 of in_1. In the last iteration step, the global value g is generated as the logical OR of

the partial global value $g^{'}$, b_3, and b_4 obtained for the nodes in_3 and in_4, respectively.

Since the form of each bit stream for intermediate values changes from protocol to protocol based on the encoding rules, in the following definition we do not specify how a bit stream is generated from a pair of an intermediate value node, and its address. Instead, we use the definition to specify the notation $\mathcal{G}(in_j, adr_j)$ as the function generating the bit streams.

Figure 2.6 An Example for Assembling Transformation



Definition 2.5.4 : Let in_j be a node of the intermediate value i and adr_j be the address for the initial functional unit of in_j. Then the function which generates a partial global value can be defined as $\mathcal{G}(in_j, adr_j) = b_j$ where b_j is the bit stream obtained for in_j.

Now, we can define the function which generates pairs of intermediate value nodes and addresses for its initial functional unit.

Definition 2.5.5 : Let in_j be a node of the intermediate value i, and adr_j be the address for the initial functional unit of in_j. Then the function which generates pairs of intermediate value nodes and addresses for its initial functional unit can be defined as :

$$\mathcal{IA}(in_j, adr_j) = \begin{cases} \{(in_{lc}, adr_{lc}), (in_{rs}, adr_{rs})\} & pre_5(in_j) \wedge pre_6(in_j) \\ \{(in_{lc}, adr_{lc})\} & pre_5(in_j) \wedge (\neg pre_6(in_j)) \\ \{(in_{rs}, adr_{rs})\} & (\neg pre_5(in_j)) \wedge pre_6(in_j) \\ \emptyset & (\neg pre_5(in_j)) \wedge (\neg pre_6(in_j)) \end{cases}$$

where adr_{lc}, and adr_{rs} are the addresses for initial functional units of leftmost child in_{lc} and right sibling in_{rs} of in_j.

Based on the previous definitions, we can give the definition for assembling transformations for each intermediate value tree node.

Definition 2.5.6 : Let $i = in_1, \ldots, in_n$ be an intermediate value and in_j be a node of i. The assembling transformation for in_j is defined as

$$\mathbf{A}(in_j) = <\mathcal{G}(in_j, adr_j), \mathcal{IA}(in_j, adr_j)>$$

In algorithm 2.4, set IA_0 is used as the seed set of the computation where $IA_0 = \{(in_1, adr_1)\}$. At each iteration step of the algorithm, bit streams for intermediate value nodes are computed and written into the partial global value. Also, at each iteration step, pairs of intermediate value nodes and addresses to be processed at the next iteration step are computed. Algorithm 2.4 stops when the necessary functional units for all intermediate value nodes are written into the global value.

Algorithm 2.4 :

Step1. $U = IA_0, V = \emptyset, g = \epsilon$.

Step2. $\forall (in_j, adr_j) \in U$ **do**

{Compute $\mathcal{G}(in_j, adr_j)$.

$g = g \,||\, \mathcal{G}(in_j, adr_j)$

Compute $\mathcal{IA}(in_j, adr_j)$.

$V = V \bigcup \mathcal{IA}(in_j, adr_j)$. }

Step3. $U = V, V = \emptyset$.

Step4. **If** $U \neq \emptyset$ **then** {Goto Step 2.} **else** {Stop.}

Now, having discussed the algorithms for all the phases of PDU encoding and decoding, we can describe a distributed model for implementing these algorithms.

Chapter 3 DISTRIBUTED IMPLEMENTATION

In this chapter, we will explain the distributed implementation model for phase algorithms of encoding and decoding transformations. This unique model is formally described through steps of refinements for different characteristics of the distributed implementation model.

Based on the formal description, the correctness and complexity analysis of the distributed implementations for the phase algorithms are presented.

3.1 Communicating Sequential Processes

The basis of the formalism for the distributed implementation model descriptions is Communicating Sequential Processes (CSP) which is a language of communicating processes. It is based on the primitive notions of concurrency and communication [Hoa 85]. In this section, the basic notions of CSP language, definitions for networks of communicating sequential processes, and rules for refinements of communicating sequential processes are given.

3.1.1 Processes

A CSP *process* is the behaviour pattern of an object. Words in uppercase letters are used to denote specific processes, and P, Q, R, etc. are used for arbitrary processes.

The basic actions of processes are called *events*. Words in lowercase letters are used to denote distinct events. Each process is associated with an *alphabet* which is the set of events of the process. The alphabet of process P is denoted by αP.

Let x be an event and P be a process. The process (x→P) is the description of an object whose first event is x and which then acts as described by P, and $\alpha(x \rightarrow P) = \alpha P \cup \{x\}$. If a process P is defined as $P = (x \rightarrow P)$, then P is said to be a *recursive* process. A process description which begins with a prefix is called *guarded.*

Let x, and y be distinct events and P, and Q distinct processes. The process (x→P|y→Q) is the description of an object whose initial event is either x or y. Depending on the initial event, the subsequent behaviour of the object is either P or Q, and $\alpha(x \rightarrow P|y \rightarrow Q) = \alpha P \cup \alpha Q \cup \{x, y\}$. In general, if B is a set of events and P(x) defines a process for each x in B, then $(x : B \rightarrow P(x))$ describes a process whose first event is any event y in B, and then behaves like P(y).

The recursion technique for the description of a single process is generalized as a set of mutually recursive descriptions where all the right-hand sides of the descriptions are guarded and each process is described once. Let x, y, and z be distinct events and P, and Q distinct processes.

$$P = (x \rightarrow P|y \rightarrow Q)$$

$$Q = (y \rightarrow Q|z \rightarrow P)$$

describes an object acting as P or Q depending on the events taking place and $\alpha P = \alpha Q = \{x, y, z\}$. Indexed variables can also be used to specify either infinite sets or large number of descriptions.

3.1.2 Concurrency and Concealment

If P and Q are distinct processes, P||Q denotes the process which behaves like a system composed of P and Q which are concurrently evolving and interacting with each other and $\alpha(P\|Q) = \alpha P \cup \alpha Q$. || is symmetric and associative, therefore $\|_{i=1}^{n} P_i$ can be used to denote parallel composition of processes, P_1,...,P_n.

The alphabet of a process contains only the relevant events whose occurrences require simultaneous participation of an environment. Therefore, it is possible to conceal internal events of a process such that they are not observed or controlled by its environment. If C is a set of events to be concealed in the process P, then P\C is a process whose behaviour is the same as that of P except that every occurrence of any event in C is concealed. Obviously, $\alpha(P\backslash C) = (\alpha P) - C$.

3.1.3 Nondeterminism

If P and Q are distinct processes, $P \sqcap Q$ denotes the process which behaves either like P or Q, where the selection is arbitrary. $\sqcap$ is idempotent, symmetric, and associative, therefore $\sqcap_{i=1}^{n} P_i$ can be used to denote the nondeterministic or of processes, P_1,...,P_n.

If P and Q are distinct processes, $P [] Q$ denotes the process which behaves either like P or Q depending on the first action. If the first action is not a possible first action of P (Q), then Q (P) will be selected. If the first action is possible for both P and Q, then it becomes equivalent to nondeterministic or. [] is idempotent, symmetric, and associative, therefore $[]_{i=1}^{n} P_i$ can be used to denote the general choice of processes, P_1,...,P_n.

3.1.4 Traces

A *trace* of a process is a finite sequence of events which the process has performed up to some moment in time. <> denotes the empty trace, <a> denotes the trace consisting of the single event a, s.t denotes the concatenation of s and t, $s \uparrow A$ denotes the trace obtained from s by deleting all events not in the set A. If A is a set of symbols, A^* is used to denote the set of all finite traces which are formed from symbols in A.

A partial ordering relation $\leq$ is defined for traces such that $s \leq t = (\exists u.\ s.u = t)$ where s is said to be a *prefix* of t. The length of the trace t is denoted by #t whereas t↓x defines the number of occurrences of the symbol x in the trace t. If s is a nonempty

sequence, its first symbol is denoted by s_0. For $1 \leq i \leq \#s$, s[i] denotes the i^{th} event in the trace s.

traces(P) denotes the set of traces of process P where $traces(P) \subseteq (\alpha P)^*$. *initials*(P) denotes the set of initial events of traces of process P where $initials(P) \subseteq (\alpha P)$. If s is a trace of P, then P/s (P *after* s) is used to denote the process whose behaviour is the P's subsequent behaviour after first engaging in all the actions recorded in s. Unless s is a trace of P, P/s is undefined.

3.1.5 Refusals, Failures and Divergences

A *refusal* of a process is a set of events all of which it may be unable to perform. In other words if X is the set of events which are offered by the environment of process P and if it is possible for P to be unable to act on its first step, then X is a refusal of P. *refusals*(P) is the set of P's initial refusals and $refusals(P) \subseteq \wp(\alpha P)$ where $\wp(\alpha P)$ is the power set of αP.

The *failures* of a process is defined as a set of pairs
$failures(P) = \{(s, X) \,|\, s \in traces(P) \wedge X \in refusals(P/s)\}$ where
$failures(P) \subseteq (\alpha P)^* \times \wp(\alpha P)$. If (s,X) is a failure of P, then P may refuse all the events in X immediately after engaging in the events recorded by s.

A *divergence* of a process is any trace of the process after which the process becomes able to perform an unbounded number of hidden internal actions without communicating to its environment. The set of all such divergences of P is denoted by *divergences*(P) where $divergences(P) \subseteq traces(P)$.

3.1.6 Communication

A *communication* is an event that is described by a pair c.v where c is the name of

the channel where the communication occurs and v is the value of the message which is communicated. The set of all messages which P can communicate on c is defined as $\alpha c(P) = \{v | c.v \in \alpha P\}$. The functions extracting channel and message components of a communication are $channel(c.v) = c$ and $message(c.v) = v$.

Let $v \in \alpha c(P)$. A process which first outputs v on c and then acts like P is defined by $(c!v \rightarrow P) = (c.v \rightarrow P)$. A process which is initially ready to input any value x on c, and then acts like P(x) is defined by $(c?x \rightarrow P(x)) = (y : \{y | channel(y) = c\} \rightarrow P(message(y)))$.

Let P and Q be processes, and c be a channel used for output by P and for input by Q. In P||Q, a communication event c.v can take place only when both processes engage simultaneously in that event. In other words, c.v occurs whenever P outputs v on c and Q simultaneously inputs the same value.

3.1.7 Communicating Processes Networks

A *network* is a parallel combination of processes. An indexed tuple notation $< P_i | 1 \leq i \leq n >$ is used to denote a network of n processes. The processes in a network may be built by parallel composition.

Networks can be either *static* in which the number of processes and their alphabets are fixed or *dynamic* in which such characteristics may change during the execution of the system.

The communication topology of a network is represented by the *communication graph.*

Definition 3.1.1 : The communication graph of the network $V =< P_i | 1 \leq i \leq n >$ is an undirected graph whose nodes represent P_i's and there is an arc between the node for P_i and that for P_j iff $\alpha P_i \cap \alpha P_j \neq \emptyset$.

As in the original CSP language [Hoa 78], all communications are assumed to be two-way communications. In other words, parallel compositions are triple-disjoint such that no event is included in the alphabets of more than two processes. Based on this restriction, the *vocabulary* of a network is defined as the events common to the alphabets of two processes.

Definition 3.1.2 : The vocabulary of the network $V =< P_i | 1 \leq i \leq n >$ is the set $\bigcup_{i=1,j=i+1}^{n} (\alpha P_i \cap \alpha P_j)$.

W is a *subnetwork* of V and it is obtained from V by removing processes in a (possibly empty) set. The communication graph of the subnetwork W of the network V is obtained by removing nodes corresponding to removed processes and arcs connected to these nodes. The vocabulary of the subnetwork W of the network V is a subset of the vocabulary of V.

The behaviour of a network $V =< P_i | 1 \leq i \leq n >$ is defined as $\|_{i=1}^{n} P_i$. The alphabet of this network V is defined as $\alpha V = \bigcup_{i=1}^{n} \alpha P_i$. The failures of the network V is defined as

$$failures(\|_{i=1}^{n} P_i) = \left\{ \left(s, \bigcup_{i=1}^{n} X_i \right) | s \in \left(\bigcup_{i=1}^{n} \alpha P_i \right)^* \bigwedge_{i=1}^{n} (s \uparrow \alpha P_i, X_i) \in failures(P_i) \right\}$$

assuming that all the processes of V are divergence-free.

The communication graph of a network captures the static essence of the network. In order to specify the dynamic nature of the network, the notion of *state* is used.

Definition 3.1.3 : A state of the network $V =< P_i | 1 \leq i \leq n >$ consists of a trace s of V together with an indexed tuple $< X_1, ..., X_n >$ of refusal sets X_i such that for each i, $(s \uparrow \alpha P_i, X_i) \in failures(P_i)$.

The initial state $\sigma_{V,i}$ of a network $V =< P_i | 1 \leq i \leq n >$ is $\sigma_{V,i} = (<>, < refusals(P_1), ..., refusals(P_n) >)$.

When V is in state (s,<X_1,...,X_n>), each process P_i becomes capable of doing any event from $\alpha P_i - X_i$ at the next step after performing $s \uparrow \alpha P_i$.

When the network $V =< P_i | 1 \leq i \leq n >$ is at state $\sigma_V = (s, < X_1, ..., X_n >)$ and event a takes place between two processes P_i and P_j, the next state of V is $\rho_V = N_{V,a}(\sigma_V) = (s.<a>, <X_1, ..., refusals(P_i/s.<a>), ..., refusals(P_j/s.<a>), ..., X_n)$. At state σ_V, the trace of V is denoted by $\sigma_V.s$ and refusal sets of V are denoted collectively by $\sigma_V.X$.

In order to specify the communications which take place at every state, the notion of *request* is used.

Definition 3.1.4 : Let $\sigma = (s, < X_1, ..., X_n >)$ be a state of the network $V =< P_i | 1 \leq i \leq n >$. A pair of distinct processes (P_i,P_j) is a request at state σ if $(\alpha P_i - X_i) \cap \alpha P_j \neq \emptyset$ and is denoted by $P_i \rightarrow^{\sigma} P_j$.

A request (P_i,P_j) at state σ is a *strong request* if $P_j \supseteq (\alpha P_i - X_i)$, and is denoted by $P_i \Rightarrow^{\sigma} P_j$. A (strong) request ($P_i$,$P_j$) at state σ is an *ungranted (strong) request* if $(\alpha P_i - X_i) \cap (\alpha P_j - X_j) = \emptyset$, and is denoted by $P_i \rightarrow^{\sigma} \bullet P_j$ ($P_i \Rightarrow^{\sigma} \bullet P_j$). A (strong) request ($P_i$,$P_j$) at state σ is a (strong) request with respect to Λ if $(\alpha P_i - X_i) \cup (\alpha P_j - X_j) \subseteq \Lambda$, and is denoted by $P_i \rightarrow^{\sigma,\Lambda} P_j$ ($P_i \Rightarrow^{\sigma,\Lambda} P_j$), where Λ is the vocabulary of the network containing P_i and P_j.

Definition 3.1.5 : A process P_i of the network $V =< P_i | 1 \leq i \leq n >$ with the vocabulary Λ is said to be *blocked* in the state σ when

- $\exists P_j.\ P_i \rightarrow^{\sigma} P_j$,
- $P_i \rightarrow^{\sigma,\Lambda} \bullet P_k$ whenever $P_i \rightarrow^{\sigma} P_k$.

We will see later, when examining the deadlock properties of the implementation model, that when all deadlock-free processes of a network are blocked at some state, the network becomes deadlocked.

In order to collectively specify the dynamic nature of all the processes of a network at each state, a graphical representation can be used.

Definition 3.1.6 : The *snapshot graph* of the network $V =< P_i | 1 \leq i \leq n >$ in the state σ is the directed graph whose nodes represent P_i's and there is a directed arc from the node for P_i to that for P_j if $<P_i,P_j>$ is a request in the state σ.

Definition 3.1.7 : Let σ be a state of the network $V =< P_i | 1 \leq i \leq n >$. A sequence of requests $< (P_{i_0}, P_{i_1}), ..., (P_{i_{r-1}}, P_{i_0}) >$ with $r \geq 3$ is defined as a *cycle of requests* in σ if, for each j, $(P_{i_j}, P_{i_{j+1}})$ is a request in σ (where addition is modulo r).

A cycle is proper if all the request targets are distinct. The length of a cycle is r. When a cycle is proper, then the number of processes involved in the requests becomes r. As can be seen from the definition 3.1.7, the minimum length for a cycle is 3; since a cycle of length 2 occurs during the interaction of two processes and is not related with the global behaviour. Instead, cycles of length 2 are defined as conflicts as follows.

Definition 3.1.8 : A state σ of the process pair <P_i,P_j> is a Γ–*conflict* if $P_i \rightarrow^{\sigma,\Gamma} \bullet P_j \wedge P_j \rightarrow^{\sigma,\Gamma} \bullet P_i$. The state is a *strong* Γ–*conflict* if at least one of the requests is strong.

Definition 3.1.9 : A process pair <P_i,P_j> is *free of* Γ–*conflict* if none of its states is a Γ–conflict. A process pair <P_i,P_j> is *free of strong* Γ–*conflict* if none of its states is a strong Γ–conflict.

Definition 3.1.10 : A network $V =< P_i | 1 \leq i \leq n >$ is conflict-free if and only if for all $i \neq j$, the pair <P_i,P_j> is conflict-free with respect to the vocabulary Λ of V. The network is free of strong conflict if and only if each pair is free of strong Λ–conflict.

3.1.8 Refinement of Communicating Sequential Processes

The development of a distributed implementation model involves a series of steps starting with an initial formal specification of the system. In a typical specification, a system is specified by an abstract machine which captures the entire functionality of the system. The notation for the specification, e.g. CSP, expresses features of the system such as concurrency, nondeterminism, etc.

In the design phase, the abstract machine is converted into a more concrete machine which embodies all the relevant properties of the previous design. This technique is named as *refinement.*

Criteria for refinement guarantees that each refining design implements the original design such that there is a correspondance between the system state spaces of two designs. In [Jif 89], the notions of *downward simulation* and *upward simulation* are described as refinement criteria. Since downward simulation is enough to explain the refinement of communicating sequential processes, we will only give the definition for downward

simulation. Before the definition for downward simulation, let us give the definition for refinement between communicating sequential processes.

Definition 3.1.11 : The process P refines Q if $\alpha P = \alpha Q$ and $failures(P) \subseteq failures(Q)$.

The notation $P \sqsupseteq Q$ is used to denote that P is a refinement of Q. Based on the definition 3.1.11, it can be said that process P is equal to P or better in the sense that it is less likely to fail. The criterion for refinement, downward simulation, is defined in the following definition.

Definition 3.1.12 : A predicate $dn : \Sigma_Q x \Sigma_P$ is called a downward simulation if it satisfies

$$dn((<>,refusals(Q)),(<>,refusals(P)))$$

$$\forall a \in (\alpha Q) \exists \sigma_P \bullet dn(\sigma_Q,\sigma_P) \wedge \rho_P = N_{P,a}(\sigma_P) \Rightarrow \exists \rho_Q \bullet \rho_Q = N_{Q,a}(\sigma_Q) \wedge dn(\rho_Q,\rho_P)$$

$$\forall c?x \in (\alpha Q - \sigma_Q.X) \wedge dn(\sigma_Q,\sigma_P) \Rightarrow c?x \in (\alpha P - \sigma_P.X)$$

$$\forall d!m \in (\alpha Q - \sigma_Q.X) \wedge dn(\sigma_Q,\sigma_P) \Rightarrow \exists e!n \bullet e!n \in (\alpha P - \sigma_P.X)$$

The first condition of definition 3.1.12 relates the initial states of networks. The second condition specifies that P at any state behaves like Q would behave at the related state. The third condition specifies that P is ready to accept any input which is agreed to by Q. The fourth condition states that if Q is ready to send messages to its environment then P must also be ready to send some messages.

The notation $dn : Q \geq P$ is used to denote that dn is a downward simulation between Q and P. The soundness of the downward simulation for process refinement is given in the following theorem.

Theorem 3.1 : If $dn : Q \geq P$ then $P \sqsupseteq Q$.

Proof : [Jif 89].

Based on the composition property of downward simulation on the concurrency operator of CSP, we can give the definition for refinement for networks of communicating sequential processes.

Definition 3.1.13 : The network $V_2 =< P_i | 1 \leq i \leq n >$ refines $V_1 =< Q_i | 1 \leq i \leq n >$ if $\forall i \ 1 \leq i \leq n \bullet P_i \sqsupseteq Q_i$.

In the definitions, 3.1.11–3.1.13, is is assumed that refining and refined processes have the same alphabet. However, during a stepwise design process a more concrete implementation usually includes some internal events while using the alphabet of the more abstract implementation as the external event set. The following definition describes this type of refinement.

Definition 3.1.14 : The process P is a *hiding refinement* of Q if $\alpha Q \subseteq \alpha P$ and $P \setminus (\alpha P - \alpha Q) \sqsupseteq Q$.

Since, the internal events are hidden in the more concrete implementation, an internal event should correspond to no state change in the more abstract implementation. There should be no divergence due to hidden internal events. Based on these requirements, downward simulation for hiding refinement is defined as follows.

Definition 3.1.15 : A predicate $dn : \Sigma_Q x \Sigma_P$ is called a downward simulation for the hiding refinement with respect to the set of internal events $\alpha P - \alpha Q$ when $\alpha Q \subseteq \alpha P$ if it satisfies

$dn((<>,refusals(Q)),(<>,refusals(P)))$

$\forall a \in (\alpha Q) \exists \sigma_P \bullet dn(\sigma_Q,\sigma_P) \wedge \rho_P = N_{P,a}(\sigma_P) \Rightarrow \exists \rho_Q \bullet \rho_Q = N_{Q,a}(\sigma_Q) \wedge dn(\rho_Q,\rho_P)$

$\forall b \in (\alpha P - \alpha Q) \exists \sigma_P \bullet dn(\sigma_Q,\sigma_P) \wedge \rho_P = N_{P,b}(\sigma_P) \Rightarrow dn(\sigma_Q,\rho_P)$

$\forall \sigma_Q \forall c?x \in (\alpha Q - \sigma_Q.X) \bullet dn(\sigma_Q,\sigma_P) \wedge \sigma_P.X \subseteq (\alpha P - \alpha Q) \Rightarrow c?x \in (\alpha P - \sigma_P.X)$

$\forall \sigma_Q \forall d!m \in (\alpha Q - \sigma_Q.X) \bullet dn(\sigma_Q,\sigma_P) \wedge \sigma_P.X \subseteq (\alpha P - \alpha Q) \Rightarrow \exists e!n \bullet e!n \in (\alpha P - \sigma_P.X)$

$\forall \sigma_P \exists k \forall s \in (\alpha P - \alpha Q)^* (\#s \geq k \Rightarrow (\alpha P - \alpha Q) \subseteq N_{P,s}(\sigma_P).X)$

The first two conditions are the same as in definition 3.1.12. The third condition states that corresponding to an internal event in P, no event takes place in Q. The fourth condition states that if Q is ready to accept an input, then P is also ready to accept the same input which happens as an external event. The fifth condition specifies that if Q is ready to send a message then P is also ready to send some message which happens as an external event. The last condition states that no infinite sequence of internal events takes place in P.

Similar to theorem 3.1, a theorem for soundness of downward simulation for hiding refinement is given in [Jif 89]. We will not repeat this theorem.

According to the fourth condition of definition 3.1.15, P should accept whatever Q accepts. However, when process renaming is necessary during a hiding refinement step, an original process Q may be refined by a renamed version of P with its new events concealed : $f^{-1}P \setminus (\alpha(f^{-1}P) - \alpha Q) \sqsupseteq Q$ where f is the injection which maps the alphabet of P onto a set of symbols A, $f : \alpha P \rightarrow A$.

Based on the downward simulation for hiding refinement, the implementation model for phase algorithms is explained in the next section.

3.2 Model

The implementation model for phases of decoding and encoding transformations is based on implementing step 2 of algorithms in spatially distributed components. Each component is used to implement a *process* which communicates with other processes through *message-passing*. Processes themselves are sequential in nature and their collection represents the overall computational activity.

In order to distinguish the functionalities of different components, distinct names are used. As an example the components where the computation for step 2 of phase algorithms are performed, are called *execution units* (EUs).

Message-passing distributed systems are classified according to four different criteria : network topology, synchrony, failure, and message buffering [Lam 90]. In the following, we will define the class of the implementation system according to the requirements of distributed implementation of algorithms and give the formal descriptions through refinements.

3.2.1 Synchrony

In all algorithms explained in Chapter 2, the processing requirement associated with each element at step 2 may be different. Hence, the entire processing is separated into many threads each continuing on different parts of the trees or the input string. Therefore, instead of synchronizing components (execution units) for unit transformations according to the slowest transformation at each step, the *completely asynchronous* model is chosen.

According to the completely asynchronous model, the messages are assumed to be eventually delivered and processes eventually respond without any upper bound on the message transmission time or process response time. This model is chosen to remove the concept of the timer.

3.2.2 Network Topology

The communication network is selected to be fixed in which, for each process, the set of directly communicatable processes, in turn components, is constant. However, the algorithms are *dynamic* in the sense that execution units receive their inputs – elements to be computed at step 2 – interactively when computations for preceeding elements are finished at other execution units.

In dynamic algorithms, a fundamental problem is the detection of termination where a process may be either active or idle. A dynamic algorithm terminates when all processes are idle and there is no message in transit. A simple solution for this problem exists when processing starts and ends at a designated controlling process [Dij 80]. Assume that such a control process (implemented on a central controller) is used to distribute processing activities for each element at step 2 among execution units. In this case the central controller (CC) knows whether an execution unit is active or idle provided that execution units which become idle send a notification message to the central controller.

The central controller keeps a status word (SW) to track the state of each execution unit. The status word itself can be specified as the combination of n concurrent processes, i.e. status bits, where each process (except the first and the n^{th}) communicate with its two adjacent status bits. Initially, all the execution units are idle. The main central controller process sends a **get** message to the first status bit. If a status bit shows an idle execution unit, then it sends a **ready** message to the main central controller process and changes the corresponding execution unit status to active. If a status bit shows an active execution unit, then it passes the **get** message to the next status bit. The assumption made here is that there is always an available execution unit. In other words, the **get** message never reaches to the n^{th} status bit showing the n^{th} execution unit as active. Once an execution unit becomes idle, the main central controller process sends a **release** message to its

status bit to change it to idle again. The termination of the algorithm is detected when all the execution units become idle again. For the termination detection, the main central controller process sends a **check** signal to the first status bit. If a status bit shows an idle execution unit, then it passes the **check** message to the next status bit. Otherwise, it sends an **active** message to the main central controller. When the **check** message reaches the n^{th} status bit, if it also shows an idle execution unit, it sends an **allidle** message to the main central controller process to specify the termination of the computation.

The main central controller process (CC_m) communicates with the external environment to input the computation requests and to output the results through **req**uest and **res**ponse messages, respectively. When it gets an external request, it sends a **start** message to the first execution unit for the computation. When an execution unit receives a **start** message, it begins the execution of step 2 of phase algorithms on parts of the input. The execution for each node is described by the local event **eu.i.compute**. When it finds further parts of the input, it sends them to the central controller using the message **new**. In the following initial description, it is assumed that in algorithms 2.1–2.4, the set obtained at the end of each unit computation contains at most two elements. These elements are processed at the next step of the algorithm. If there are two elements, the execution unit sends one of the elements in a **new** message. If there is a single element, the execution unit does not send any message to the central controller. When it finishes its computation, i.e. there is no element to be processed, it notifies the central controller with a **completed** message. The central controller finds an idle execution unit for each **new** message and changes the status of the execution unit for each **completed** message. Based on these assumptions, the implementation is described as the combination of the central controller and n concurrent EUs.

Initial Description

$IMP=CC\|(\|_{i=1}^{n} EU_i)$

$CC=CC_m\|SW$

$CC_m=(ext?req\rightarrow sw.1!get\rightarrow sw.1?ready\rightarrow eu.1!start\rightarrow P_{CC})$

$P_{CC}=\Box_{i=1}^{n}((eu.i?new\rightarrow R_{CC})\Box(eu.i?completed\rightarrow S_{CC}))$

$R_{CC}=(sw.1!get\rightarrow\Box_{i=1}^{n}(sw.i?ready\rightarrow eu.i!start)\rightarrow P_{CC})$

$S_{CC}=(sw.i!release\rightarrow sw.1!check\rightarrow(\Box_{i=1}^{n}(sw.i?active\rightarrow P_{CC})\Box(sw.n?allidle\rightarrow ext!res\rightarrow CC_m)))$

$SW=\|_{i=1}^{n} I_i$

$I_i=(sw.i?get\rightarrow sw.i!ready\rightarrow A_i)\Box(sw.i?check\rightarrow sw.i+1!check\rightarrow I_i)\ 1\leq i<n$

$I_n=(sw.n?get\rightarrow sw.n!ready\rightarrow A_n)\Box(sw.n?check\rightarrow sw.n!allidle\rightarrow I_n)$

$A_i=(sw.i?get\rightarrow sw.i+1!get\rightarrow A_i)\Box(sw.i?release\rightarrow I_i)\Box(sw.i?check\rightarrow sw.i!active\rightarrow A_i)\ 1\leq i<n$

$A_n=(sw.n?release\rightarrow I_n)\Box(sw.n?check\rightarrow sw.n!active\rightarrow A_n)$

$EU_i=(eu.i?start\rightarrow AEU_i)$

$AEU_i=(eu.i.compute\rightarrow(AEU_i\sqcap BEU_i))$

$BEU_i=(eu.i!new\rightarrow AEU_i)\sqcap(eu.i!completed\rightarrow EU_i)$

Input and output for each computation at step 2 are also carried through messages between execution units and an interface controller (IC) on which the process to read from and write to a common memory is run. The messages involved in the communication between the interface controller and execution units are **read, write** and **data**. Refinement 3.1 describes the implementation as a collection of two controllers and n execution units. It should be noted in refinement 3.1 and all the following refinements, for the processes which are not redescribed, the previous description is used.

Refinement 3.1

$IMP=CC\|(\|_{i=1}^{n} EU_i)\|IC$

$EU_i=(eu.i?start\rightarrow AEU_i)$

$AEU_i=(eu.i!read\rightarrow eu.i?data\rightarrow eu.i.compute\rightarrow eu.i!write\rightarrow(AEU_i \sqcap BEU_i))$

$BEU_i=(eu.i!new\rightarrow AEU_i)\sqcap(eu.i!completed\rightarrow EU_i)$

$IC=\square_{i=1}^{n}((eu.i?read\rightarrow eu.i!data\rightarrow IC)\square(eu.i?write\rightarrow IC))$

In order to prove that the IMP of refinement 3.1, i.e. $IMP_{3.1}$ is a hiding refinement of the initial IMP description, IMP_{id}, we need to show EU_i's of refinement 3.1, i.e. $EU_{i|3.1}$ are hiding refinements of EU_i's of the initial description, i.e.. $EU_{i|id}$

Lemma 3.1 : $\forall i\ 1 \leq i \leq n \bullet EU_{i|3.1} \setminus (\alpha EU_{i|3.1} - \alpha EU_{i|id}) \sqsupseteq EU_i$.

Proof : In order to show it is a hiding refinement with respect to the set of internal events $\alpha EU_{i|3.1}$–$\alpha EU_{i|id}$, we have to find a downward simulation relation between $EU_{i|id}$'s and $EU_{i|3.1}$'s. In order to define this relation, we can use two-tuple variables $(n_{sc,i}(s),n_{n,i}(s))$ for a trace s of EU_i where

$n_{sc,i}(s)=(s\downarrow eu.i.start-s\downarrow eu.i.completed)$ and

$n_{n,i}(s)=s''\downarrow eu.i.new\ where\big(s=s'.<eu.i.completed>.s''\wedge s''\downarrow eu.i.completed=0\big)$. Using this two-tuple variable, the downward simulation for hiding refinement between $EU_{i|id}$'s and $EU_{i|3.1}$'s is :

$dn_1(\sigma_{id},\sigma_{3.1})=((n_{sc,i}(\sigma_{id}.s)=n_{sc,i}(\sigma_{3.1}.s)\wedge n_{n,i}(\sigma_{id}.s)=n_{n,i}(\sigma_{3.1}.s)))$ where σ_{id}, and $\sigma_{3.1}$ are used to denote states of $EU_{i|id}$'s and $EU_{i|3.1}$'s, respectively to simplify the no-

tation.

$\forall i\ 1 \le i \le n$

1. $n_{sc,i}(<>)=n_{n,i}(<>)=0 \Rightarrow dn_1((<>,refusals(IMP_{id})),(<>,refusals(IMP_{3.1})))$

2. $\exists a \in \alpha EU_{i|id} \exists \sigma_{3.1} \bullet dn_1(\sigma_{id},\sigma_{3.1}) \wedge N_{3.1,a}(\sigma_{3.1})=\rho_{3.1} \Rightarrow$

   $n_{sc,i}(\sigma_{id}.s)=n_{sc,i}(\sigma_{3.1}.s) \wedge n_{n,i}(\sigma_{id}.s)=n_{n,i}(\sigma_{3.1}.s) \Rightarrow$

   $[a=eu.i.start \Rightarrow n_{sc,i}(\rho_{3.1}.s)=1=n_{sc,i}(\sigma_{id}.s)+1 \wedge n_{n,i}(\rho_{3.1}.s)=n_{n,i}(\sigma_{3.1}.s)=n_{n,i}(\sigma_{id}.s)] \vee$

   $[a=eu.i.completed \Rightarrow n_{sc,i}(\rho_{3.1}.s)=0=n_{sc,i}(\sigma_{id}.s)-1 \wedge n_{n,i}(\rho_{3.1}.s)=0] \vee$

   $[a=eu.i.new \Rightarrow n_{sc,i}(\rho_{3.1}.s)=n_{sc,i}(\sigma_{3.1}.s)=n_{sc,i}(\sigma_{id}.s) \wedge n_{n,i}(\rho_{3.1}.s)=n_{n,i}(\sigma_{3.1}.s)+1=n_{n,i}(\sigma_{id}.s)+1] \Rightarrow$

   $\exists \rho_{id} \bullet \rho_{id}=N_{id,a}(\sigma_{id}) \wedge n_{sc,i}(\rho_{id}.s)=n_{sc,i}(\rho_{3.1}.s) \wedge n_{n,i}(\rho_{id}.s)=n_{n,i}(\rho_{3.1}.s) \Rightarrow$

   $\exists \rho_{id} \bullet \rho_{id}=N_{id,a}(\sigma_{id}) \wedge dn_1(\rho_{id},\rho_{3.1})$

3. $\exists b \in (\alpha EU_{i|3.1}-\alpha EU_{i|id}) \exists \sigma_{3.1} \bullet dn_1(\sigma_{id},\sigma_{3.1}) \wedge N_{3.1,a}(\sigma_{3.1})=\rho_{3.1} \Rightarrow$

   $n_{sc,i}(\sigma_{id}.s)=n_{sc,i}(\sigma_{3.1}.s) \wedge n_{n,i}(\sigma_{id}.s)=n_{n,i}(\sigma_{3.1}.s) \Rightarrow$

   $[n_{sc,i}(\rho_{3.1}.s)=n_{sc,i}(\sigma_{3.1}.s)=n_{sc,i}(\sigma_{id}.s) \wedge n_{n,i}(\rho_{3.1}.s)=n_{n,i}(\sigma_{3.1}.s)=n_{n,i}(\sigma_{id}.s)] \Rightarrow dn_1(\sigma_{id},\rho_{3.1})$

4. $\exists \sigma_{id} \exists c \in (\alpha EU_{i|id}-\sigma_{id}.X) \bullet dn_1(\sigma_{id},\sigma_{3.1}) \wedge \sigma_{3.1}.X \subseteq (\alpha EU_{i|3.1}-\alpha EU_{i|id}) \Rightarrow$

   $\alpha EU_{i|id} \subseteq (\alpha EU_{i|3.1}-\sigma_{3.1}.X) \wedge (\alpha EU_{i|id}-\sigma_{id}.X) \subseteq \alpha EU_{i|id} \Rightarrow$

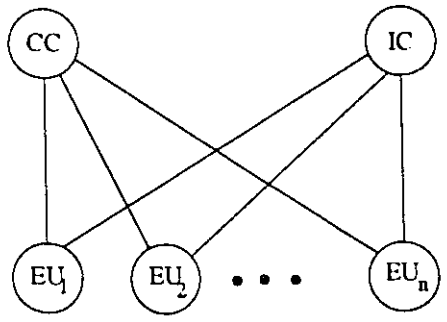
   $(\alpha EU_{i|id}-\sigma_{id}.X) \subseteq (\alpha EU_{i|3.1}-\sigma_{3.1}.X) \Rightarrow c \in (\alpha EU_{i|3.1}-\tau_{3.1}.X)$

5. $\forall \sigma_{3.1} \exists k \forall s \in (\alpha EU_{i|3.1}-\alpha EU_{i|id})^{*} (\alpha EU_{i|3.1}-N_{3.1,s}(\sigma_{3.1}).X=\{eu.i.new,eu.i.completed\})$

[]

The network topology for distributed implementation becomes a double star network where the CC and the IC are connected to all EUs, and its communication graph is shown in Figure 3.1. It is an undirected graph where an arc joining two processes (components) means that each can send messages to the other.

Figure 3.1 Communication Graph for IMP in Refinement 3.1



The specification for the central controller in the initial description, is based on the assumption that there is always an available execution unit for new processing requests. If we remove this assumption, a queueing mechanism becomes necessary to store the **new** messages when there is no idle execution unit. When a **new** message comes the main central controller process sends a **get** message to the first status bit. If all the bits show active execution units, the n^{th} status bit process sends a **noava**ilable message to the main central controller process which sends a **put** message to the queue (QUE) to store the **new** message. Each time an execution unit sends a **completed** message, the main central controller checks the queue using a **take** message. If there is no message in the queue, the queue replies with an **empty** message and then the status of the execution unit which sent the **completed** message is turned to idle. If the queue is not empty, then it replies with a **qdata** message which is sent to the execution unit which sent the **completed** message. The size of the queue is assumed to be sufficiently large so that there is always an available place for a **new** message. Based on these assumptions, refinement 3.2 shows the central controller as a combination of the main process, the status word and the queue.

Refinement 3.2

$CC{=}CC_m \| SW \| QUE_{<>}$

$R_{CC}{=}(sw.1!get{\rightarrow}(\Box_{i=1}^{n}(sw.i?ready{\rightarrow}eu.i!start)\Box(sw.n?noava{\rightarrow}queue!put)){\rightarrow}P_{CC})$

$S_{CC}{=}(queue!take{\rightarrow}((queue?empty{\rightarrow}sw.i!release{\rightarrow}Q_{CC})\Box(queue?qdata{\rightarrow}eu.i!start{\rightarrow}P_{CC})))$

$Q_{CC}{=}(sw.1!check{\rightarrow}(\Box_{i=1}^{n}(sw.i?active{\rightarrow}P_{CC})\Box(sw.n?allidle{\rightarrow}ext!res{\rightarrow}CC_m)))$

$QUE_{<>}{=}(queue?put(x){\rightarrow}QUE_{<x>})\Box(queue?take{\rightarrow}queue!empty)$

$QUE_{<x>s}{=}(queue?put(y){\rightarrow}QUE_{<x>s<y>})\Box(queue?take{\rightarrow}queue!qdata(x){\rightarrow}QUE_s)$

$A_n{=}(sw.n?get{\rightarrow}sw.n!noava{\rightarrow}A_n)\Box(sw.n?release{\rightarrow}I_n)\Box(sw.n?check{\rightarrow}sw.n!active{\rightarrow}A_n)$

Based on the description of IMP according to refinement 3.2, now we can prove that the $IMP_{3.2}$ is a hiding refinement of the $IMP_{3.1}$ by showing $CC_{3.2}$ is a hiding refinement of $CC_{3.1}$.

Lemma 3.2 : $CC_{3.2} \setminus (\alpha CC_{3.2} - \alpha CC_{3.1}) \sqsupseteq CC_{3.1}$.

Proof : In order to show it is a hiding refinement with respect to the set of internal events $\alpha CC_{3.2}$-$\alpha CC_{3.1}$, we have to find a downward simulation relation between $CC_{3.2}$ and $CC_{3.1}$. In order to define this relation, we can use n-tuple variable $(n_{rr,1}(s),...,n_{rr,n}(s))$ for a trace s of CC where $n_{rr,i}(s){=}(s{\downarrow}sw.i.ready{-}s{\downarrow}sw.i.release)$. Using this n-tuple variable, the downward simulation for hiding refinement between $CC_{3.2}$ and $CC_{3.1}$ is: $dn_2(\sigma_{3.1},\sigma_{3.2}){=}\left(\bigwedge_{i=1}^{n} n_{rr,i}(\sigma_{3.1}.s){=}n_{rr,i}(\sigma_{3.2}.s)\right)$ where $\sigma_{3.1}$, and $\sigma_{3.2}$ are used to denote states of $CC_{3.1}$ and $CC_{3.2}$, respectively. The proof of dn_2 being a downward simulation for hiding refinement is similar to that in the proof of Lemma 3.1, and is omitted. []

3.2.3 Failure

In message passing models, both process failures and communication failures may occur. Communication failures can generate loss, duplication or corruption of messages. The implementation model is assumed to be free of communication failures. This assumption is necessary since there is no clock mechanism, and thus it is not possible to detect lost messages.

If a failed process does nothing, then such a failure is called a *halting failure.* In the implementation model, a restrictive form of halting failure is permitted where the central controller and interface controller never fail and execution units fail in the sense that they become *inoperative.* In other words, once an execution unit finishes its computation correctly it sends either a **completed** message to the central controller to indicate that it becomes idle or an **inoperative** message to indicate that it becomes unable to accept new **start** messages from the central controller. Once a process becomes inoperative, it cannot be restarted again. However, the overall implementation can function until all the execution units become inoperative which results in an inoperative implementation.

In order to accommodate the changes due to the failure, the central controller should be modified. When the central controller receives an external request, it tries to find an idle and unfaulty execution unit using the status word (CC_m). If there is no such execution unit, it sends a **broken** message as the reply of the request (U_{CC}). When the main central controller process receives an **inoperative** message from an execution unit, it changes its status bit by sending a **faulty** message. Once a status bit is converted to faulty, the main central controller process sends a **check** message to the first status bit. If there is a status bit showing an active execution unit, the central controller goes back to wait for messages from the execution units. If there is no active execution unit, then the n^{th} status bit replies with a **noactive** message which means all the execution units

are either idle or inoperative. In order to distinguish different cases, the main central controller sends a **diag**nostic message to the first status bit. If there is any status bit showing an idle execution unit, it replies with an **idle** message. If all the status bits are showing inoperative execution units, the n^{th} status bit sends an **allfaulty** message to the main central controller process. The central controller and execution units are redescribed according to the failure model in refinement 3.3.

Refinement 3.3

$CC_m=(ext?req\rightarrow sw.1!get\rightarrow(\Box_{i=1}^{n}(sw.i?ready\rightarrow eu.i!start\rightarrow P_{CC})))$

$P_{CC}=\Box_{i=1}^{n}((eu.i?new\rightarrow R_{CC})\Box(eu.i?completed\rightarrow S_{CC})\Box(eu.i?inoperative\rightarrow T_{CC}))$

$Q_{CC}=(sw.1!check\rightarrow(\Box_{i=1}^{n}(sw.i?active\rightarrow P_{CC})\Box(sw.n?noactive\rightarrow$

$sw.1!diag\rightarrow(\Box_{i=1}^{n}(sw.i?idle\rightarrow ext!res\rightarrow CC_m)\Box(sw.n?allfaulty\rightarrow ext!broken\rightarrow U_{CC}))$

$T_{CC}=(sw.i!faulty\rightarrow Q_{CC})$

$U_{CC}=(ext?req\rightarrow ext!broken\rightarrow U_{CC})$

$I_i=(sw.i?get\rightarrow sw.i!ready\rightarrow A_i)\Box(sw.i?check\rightarrow sw.i+1!check\rightarrow I_i)\Box(sw.i?diag\rightarrow sw.i!idle\rightarrow I_i)\ 1\leq i<n$

$I_n=(sw.n?get\rightarrow sw.n!ready\rightarrow A_n)\Box(sw.n?check\rightarrow sw.n!noactive\rightarrow I_n)\Box(sw.n?diag\rightarrow sw.n!idle\rightarrow I_n)$

$A_i=(sw.i?get\rightarrow sw.i+1!get\rightarrow A_i)\Box(sw.i?release\rightarrow I_i)\Box(sw.i?faulty\rightarrow F_i)\Box$

$(sw.i?check\rightarrow sw.i!active\rightarrow A_i)\ 1\leq i<n$

$A_n=(sw.n?get\rightarrow sw.n!noava\rightarrow A_n)\Box(sw.n?release\rightarrow I_n)\Box(sw.n?faulty\rightarrow F_n)\Box$

$(sw.n?check\rightarrow sw.n!active\rightarrow A_n)$

$F_i=(sw.i?get\rightarrow sw.i+1!get\rightarrow F_i)\Box(sw.i?check\rightarrow sw.i+1!check\rightarrow F_i)\Box$

$(sw.i?diag\rightarrow sw.i+1!diag\rightarrow F_i)\ 1\leq i<n$

$F_n=(sw.n?get\rightarrow sw.n!noava\rightarrow F_n)\Box(sw.n?check\rightarrow sw.n!noactive\rightarrow F_n)\Box$

$(sw.n?diag\rightarrow sw.n!allfaulty\rightarrow F_n)$

$BEU_i=((eu.i!new\rightarrow AEU_i)\sqcap(eu.i!completed\rightarrow EU_i)\sqcap(eu.i!inoperative\rightarrow EU_i))$

In previous refinements 3.1 and 3.2, the refined implementation's alphabet is a subset of the refining implementation. However, according to refinement 3.3 the n^{th} status word sends a **noactive** message to process CC_m as the reply for **check** message instead of the **allidle** message of refinement 3.2. This requires the use of process renaming function f where $f^{-1} : \alpha CC_{3.3} \rightarrow \alpha CC_{3.2}$ such that

$$f^{-1}(a) = \begin{cases} sw.n.allidle & a = sw.n.noactive \\ a & a \in (\alpha CC_{3.3} - \{sw.n.noactive\}) \end{cases}$$

Lemma 3.3 :

$$a.\ \forall i\ 1 \leq i \leq n \bullet EU_{i|3.3} \setminus (\alpha EU_{i|3.3} - \alpha EU_{i|3.2}) \sqsupseteq EU_{i|3.2}$$

$$b.\ f^{-1}CC_{3.3} \setminus (\alpha(f^{-1}CC_{3.3}) - \alpha CC_{3.2}) \sqsupseteq CC_{3.2}$$

Proof :

a. Use the downward simulation dn_1 between the states of $EU_{i|3.2}$ and $EU_{i|3.3}$.

b. Use the downward simulation dn_2 between the states of $CC_{3.2}$ and $CC_{3.3}$.

Again, the proofs of dn_1 and dn_2 being downward simulations for hiding refinement are similar to that in the proof of Lemma 3.1, and are omitted. []

3.2.4 Message Buffering

Message buffering is necessary in message passing models, since there is a delay between the time a message is sent and the time it is received.

In the implementation model, the first-in first-out (FIFO) message buffering is used via a distributed buffering scheme. Two one-size buffers (BCC_i and BIC_i) are associated with each execution unit. BCC_i is used for the bidirectional communication between the execution unit and the central controller. The other one-size buffer BIC_i is used for the bidirectional communication between the execution unit and the interface controller. In order to provide the FIFO policy while either controller is serving the one-size buffers of execution units, an n–size buffer is assigned to each controller which is serving n execution units.

The n-size buffers (BUF_{CC} and BUF_{IC}) are collections of n one-size buffers ($BUFCC_i$'s and $BUFIC_i$'s). Only $BUFCC_1$ and $BUFIC_1$ are connected to execution units. In order to send a message to either controller, an execution unit first sends a **bccheck** or a **biccheck** message to BCC_i or BIC_i, respectively. If the one-size buffer is empty, it replies with a **bccempty** or **bicempty** message. Otherwise, a **bccfull** or **bicfull** message is sent. If the one-size buffer is empty, the execution unit puts the message into the buffer, and then sends a **trig**ger message to either $BUFCC_1$ or $BUFIC_1$. If the one-size buffer is full, then the execution unit repeats checking it until it becomes empty. When $BUFCC_1$ or $BUFIC_1$ receives a **trig**ger message from an execution unit EU_i, it passes the index i to the next one-size buffer. The n^{th} one-size buffer constituting the n-size buffer communicates with the controller by sending a **eun**umber message. The controller sends a **ready** message to the corresponding buffer of the execution unit whose index is received. Once the one-size buffer of the execution unit receives the **ready** message, it sends the stored message to the controller.

The messages sent from controllers, i.e. **start** and **data**, do not require this type of handshaking, since it is always guaranteed that when a **start** message from the central controller or a **data** message from the interface controller is sent to the one-size buffer of an execution unit, it is empty. Based on this assumption, implementation is given as the combination of the controllers, n-size buffers, and n execution units each coupled with two one-size buffers in refinement 3.4.

Refinement 3.4

$$IMP = CC \| BUF_{CC} \| (\|_{i=1}^{n} (BCC_{i,<>} \| EU_i \| BIC_{i,<>})) \| BUF_{IC} \| IC$$

$$CC_m = (ext?req \rightarrow sw.1!get \rightarrow (\Box_{i=1}^{n} (sw.i?ready \rightarrow bcc.i!start \rightarrow P_{CC})))$$

$$P_{CC} = (bufcc?eun \rightarrow bcc.eun!ready \rightarrow ((bcc.eun?new \rightarrow R_{CC}) \Box (bcc.eun?completed \rightarrow S_{CC}) \Box (bcc.eun?inoperative \rightarrow T_{CC})))$$

$$R_{CC} = (sw.1!get \rightarrow (\Box_{i=1}^{n} (sw.i?ready \rightarrow bcc.i!start) \Box (sw.n?noava \rightarrow queue!put)) \rightarrow P_{CC})$$

$$S_{CC} = (queue!take \rightarrow ((queue?empty \rightarrow sw.i!release \rightarrow Q_{CC}) \Box (queue?qdata \rightarrow bcc.i!start \rightarrow P_{CC})))$$

$$BUF_{CC} = \|_{i=1}^{n} BUFCC_i$$

$$BUFCC_1 = \Box_{i=1}^{n} (bufcc.1.i?trig \rightarrow bufcc.2!i \rightarrow BUFCC_1)$$

$$BUFCC_i = (bufcc.i?val \rightarrow bufcc.i+1!val \rightarrow BUFCC_i) \; 1<i<n$$

$$BUFCC_n = (bufcc.n?val \rightarrow bufcc!eun(val) \rightarrow BUFCC_n)$$

$$BCC_{i,<>} = (eu.i?bccheck \rightarrow eu.i!bccempty \rightarrow BCC_{i,<>}) \Box (bcc.i?start \rightarrow eu.i!start \rightarrow BCC_{i,<>}) \Box (eu.i?new \rightarrow BCC_{i,<new>}) \Box (eu.i?completed \rightarrow BCC_{i,<completed>}) \Box (eu.i?inoperative \rightarrow BCC_{i,<inoperative>})$$

$$BCC_{i,<x>} = (eu.i?bccheck \rightarrow eu.i!bccfull \rightarrow BCC_{i,<x>}) \Box (bcc.i?ready \rightarrow bcc.i!x \rightarrow BCC_{i,<>})$$

AEU_i=($eu.i!bicheck$→(($eu.i?bicfull$→AEU_i)□

($eu.i?bicempty$→$eu.i!read$→$bufic.1.i!trig$→$eu.i?data$→$eu.i.compute$→

$eu.i!write$→$bufic.1.i!trig$→(AEU_i⊓BEU_i))))

BEU_i=($eu.i!bccheck$→(($eu.i?bccfull$→BEU_i)□

($eu.i?bccempty$→($eu.i!new$→$bufcc.1.i!trig$→AEU_i)⊓

($eu.i!completed$→$bufcc.1.i!trig$→EU_i)⊓

($eu.i!inoperative$→$bufcc.1.i!trig$→EU_i))))

$BIC_{i,<>}$=($eu.i?bicheck$→$eu.i!bicempty$→$BIC_{i,<>}$)□($bic.i?data$→$eu.i!data$→$BIC_{i,<>}$)□

($eu.i?read$→$BIC_{i,<read>}$)□($eu.i?write$→$BIC_{i,<write>}$)

$BIC_{i,<x>}$=($eu.i?bicheck$→$eu.i!bicfull$→$BIC_{i,<x>}$)□($bic.i?ready$→$bic.i!x$→$BIC_{i,<>}$)

IC=($bufic?eun$→$bic.eun!ready$→(($bic.eun?read$→$bic.eun!data$)□($bic.eun?write$))→IC)

BUF_{IC}=$\|_{i=1}^{n}$ $BUFCC_i$

$BUFIC_1$=$\Box_{i=1}^{n}$($bufic.1.i?trig$→$bufic.2!i$→$BUFIC_1$)

$BUFIC_i$=($bufic.i?val$→$bufic.i+1!val$→$BUFIC_i$) $1<i<n$

$BUFIC_n$=($bufic.n?val$→$bufic!eun(val)$→$BUFIC_n$)

In order to show IMP of refinement 3.4 is a hiding refinement of $IMP_{3.3}$, we have to show controller components coupled with buffers are the refinements of controllers of refinement 3.3, and similarly the execution units of refinement 3.4 are refinements of those of refinement 3.3.

Lemma 3.4 :

a. $EU_{i|3.4} \backslash S_1 \supseteq EU_{i|3.3}$ where $S_1 = \alpha EU_{i|3.4} - \alpha EU_{i|3.3}$

b. $CC_{3.4} \| BUF_{CC} \| (\|_{i=1}^{n} BCC_{i,<>}) \backslash S_2 \supseteq CC_{3.3}$ where $S_2 = \alpha CC_{3.4} \cup \alpha BUF_{CC} \cup \left(\bigcup_{i=1}^{n} \alpha BCC_i \right) - \alpha CC_{3.3}$

c. $IC_{3.4} \| BUF_{IC} \| (\|_{i=1}^{n} BIC_{i,<>}) \backslash S_3 \supseteq IC_{3.3}$ where $S_3 = \alpha IC_{3.4} \cup \alpha BUF_{IC} \cup \left(\bigcup_{i=1}^{n} \alpha BIC_i \right) - \alpha IC_{3.3}$

Proof :

a. Use the downward simulation dn_1 between the states of $EU_{i|3.3}$ and $EU_{i|3.4}$.

b. Use the downward simulation dn_2 between the states of $CC_{3.3}$ and those of $CC_{3.4} \| BUF_{CC} \| (\|_{i=1}^{n} BCC_i)$.

c. In order to define the downward simulation dn_3 between the states of $IC_{3.3}$ and those of $IC_{3.4} \| BUF_{IC} \| (\|_{i=1}^{n} BIC_i)$, we can use 2–tuple variables $(n_{rd,i}(s), n_{w,i}(s))$ for each index i for a trace of refined and refining processes where

$n_{rd,i}(s) = (s \downarrow eu.i.read - s \downarrow eu.i.data)$ and $n_{w,i}(s) = s \downarrow eu.i.write$. Using these two-tuple variables, the downward simulation for hiding refinement between $IC_{3.3}$ and $IC_{3.4} \| BUF_{IC} \| (\|_{i=1}^{n} BIC_i)$ is :

$dn_3(\sigma_{3.3}, \sigma_{3.4}) = \left(\bigwedge_{i=1}^{n} (n_{rd,i}(\sigma_{3.3}.s) = n_{rd,i}(\sigma_{3.4}.s) \wedge n_{w,i}(\sigma_{3.3}.s) = n_{w,i}(\sigma_{3.4}.s)) \right)$ where $\sigma_{3.3}$, and $\sigma_{3.4}$ are used to denote the states of $IC_{3.3}$ and $IC_{3.4} \| BUF_{IC} \| (\|_{i=1}^{n} BIC_i)$, respectively. Again, the proof of dn_1, dn_2, and dn_3 being downward simulations for hiding refinement are similar to that in the proof of Lemma 3.1, and are omitted. []

3.3 Correctness

Until now, we have given the model as an abstraction of the implementation. However, we have not introduced any reasoning about the properties of implementation. In this section, the basis for the correctness of the model is discussed.

Safety and *liveness* are the properties most often investigated in distributed systems. Intuitively a safety property stipulates that something bad does not happen during execution and a liveness property stipulates that something good eventually does happen [Lam 77]. A safety or liveness property is a stipulation about an individual behaviour of the system. A system satisfies the property if it is true for all behaviours of the system.

There are a number of proof systems for concurrent programs written in CSP [Hoa 81, Sou 83, Ree 88]. Hoare introduced a *satisfies* relation in order to prove the safety properties of CSP programs. This approach is explained in subsection 3.3.1 and it is applied in subsection 3.3.2 to establish certain safety properties of the implementation model.

Liveness properties are more difficult to handle. The *satisfies* relation can not be used for proving the liveness properties since it is based on the finite past of a process. However, a liveness property is related to the happening of "good" things perhaps infinitely often. In order to establish the liveness, usually the notion of *fairness* is also needed. In subsection 3.3.2, the deadlock-freedom of IMP is shown as a safety property. Based on this property, the conflict-freedom of IMP and the fairness assumptions, the liveness properties are proven.

3.3.1 Specifications

In [Hoa 85], a *specification* of a product is given as "a description of the way it is intended to behave." Using the term process instead of product, "if P is a process which meets a specification S, we say that P *satisfies* S" and is denoted by P **sat** S.

The observation about a process P, whether it satisfies a specification, is done based on the trace of events which happen up to a certain point in time. If P **sat** S, then $\forall t.\ t \in traces(P) \Rightarrow S$.

There are a number of useful laws regarding the satisfies relation given in [Hoa 85]. The following is an incomplete list of laws about specifications used in the following section.

$L1.\ P$ snt $true$

$L2.\ (\forall n.\ P$ snt $S(n)) \Rightarrow P$ snt $(\forall n.\ S(n))$

$L3.\ (P$ snt $S \wedge (S \Rightarrow T)) \Rightarrow P$ snt T

$L4.\ (P\ sat\ S(t) \wedge Q\ sat\ T(t)) \Rightarrow (P||Q)\ sat\ (S(t \uparrow \alpha P) \wedge T(t \uparrow \alpha Q))$

3.3.2 Safety of IMP

Since algorithmic details for different algorithms for phases of encoding and decoding are excluded from the given process specifications, the properties to be proven in this section are those related with interprocess communications. In other words, in the following discussion about the system, it is assumed that computations , i.e. individual transformations, which take place in components are correct.

Safety Property (SP) 1 : CC_m does not send another **get**, **check**, or **diag**nostic message before receiving a reply for the previous one.

$$IMP \text{ snt } \left[0 \le V(t) = [\#(t \restriction \{sw.1.get, sw.1.check, sw.1.diag\}) - \#\left(t \restriction \left(\left(\bigcup_{i=1}^{n} \{sw.i.ready, sw.i.idle, sw.i.active\}\right) \cup \{sw.n.noava, sw.n.noactive, sw.n.allfaulty\}\right)\right) \le 1\right]$$

Proof : According to refinement 3.4,

CC_m snt $[(t[\#t]=sw.1.get \wedge s=t.u) \Rightarrow ((\exists i.\ 1 \le i \le n \wedge u[1]=sw.i.ready) \vee u[1]=sw.n.noava)]$

CC_m snt $[(t[\#t]=sw.1.check \wedge s=t.u) \Rightarrow ((\exists i.\ 1 \le i \le n \wedge u[1]=sw.i.active) \vee u[1]=sw.n.noactive)]$

CC_m snt $[(t[\#t]=sw.1.diag \wedge s=t.u) \Rightarrow ((\exists i.\ 1 \le i \le n \wedge u[1]=sw.i.idle) \vee u[1]=sw.n.allfaulty)]$

Therefore, if $t=t'.\langle sw.1.get \rangle \vee t=t'.\langle sw.1.check \rangle \vee t=t'.\langle sw.1.diag \rangle$, then $V(t)=V(t')+1$. Assuming CC_m satisfies the SP 1 for t', V(t') becomes 0 since V(t') can

be 1 only if the last event of t' is either one of sw.1.get, sw.1.check, or sw.1.diag. In this case the next event cannot be one of them again. Therefore, V(t) becomes 1. Similarly, if $t=t'.\langle sw.i.ready\rangle \vee t=t'.\langle sw.i.active\rangle \vee t=t'.\langle sw.i.idle\rangle \vee t=t'.\langle sw.n.noava\rangle \vee t=t'.\langle sw.n.noactive\rangle \vee t=t'.\langle sw.n.allfaulty\rangle$, then $V(t)=V(t')-1$. Assuming CC_m satisfies the SP 1 for t', V(t') becomes 1 since V(t') can be 0 only if the last event of t' is a message sent from SW to CC_m. In this case the next event cannot be sending a SW message again. Therefore, V(t) becomes 0. []

SP 2 : An execution unit is ready only if there is a **release** message for every **ready** message.

$$IMP\ snt\left[\bigwedge_{i=1}^{n}(t=t'.\langle sw.i.ready\rangle \Rightarrow \#t'\lfloor sw.i.ready=\#t'\lfloor sw.i.release)\right]$$

Proof : Assume $\#t'\lfloor sw.i.ready \neq \#t'\lfloor sw.i.release$, in this case there is either consecutive sw.i.ready events or consecutive sw.i.release events in the trace $t''\uparrow\{sw.i.ready,sw.i.release\}$. However, according to refinement 3.4, the status word process for EU_i can engage in sw.i.ready event only when it acts like I_i and after engaging in sw.i.ready event it starts acting like A_i. In order to turn to the state where it acts like I_i, the only event is sw.i.release. Therefore consecutive sw.i.ready events cannot take place in the trace $t''\uparrow\{sw.i.ready,sw.i.release\}$. Similar analysis can be done to show that consecutive sw.i.release events cannot take place in the trace $t''\uparrow\{sw.i.ready,sw.i.release\}$. Contradiction. []

SP 3 : An idle or inoperative execution unit can not send any more messages.

$$IMP\ snt\left[\bigwedge_{i=1}^{n}(0\leq V(t)=[t\downarrow eu.i.start-(t\lfloor eu.i.completed+t\lfloor eu.i.inoperative)])\right]$$

Proof : Assume $t=t'.\langle eu.i.start\rangle$ is a trace of EU_i which satisfies the SP 3 for t'. Then EU_i also satisfies the SP 3 for t, since $V(t)=V(t')+1$. Assume $t=t'.\langle eu.i.completed\rangle \vee t=t'.\langle eu.i.inoperative\rangle$ is a trace of EU_i which satisfies the SP

3 for t'. Then EU_i also satisfies the SP 3 for t, if $V(t')\geq 1$. Assume $V(t')=0$, then the last event of the trace $t'\restriction\{eu.i.start,eu.i.completed,eu.i.inoperative\}$ should be either eu.i.completed or eu.i.inoperative. According to refinement 3.4, after sending a **completed** or an **inoperative** message the process waits for a **start** message. This completes the contradiction, and therefore $V(t')\geq 1$. []

SP 4 : An active execution unit can not be started.

$$IMP\,snt\left[\bigwedge_{i=1}^{n}(V(t)=[t\downarrow bcc.i.start-(t\downarrow bcc.i.completed+t\downarrow bcc.i.inoperative)]\leq 1)\right]$$

Proof : Assume $t=t'.<bcc.i.completed>\vee t=t'.<bcc.i.inoperative>$ is a trace of CC_m which satisfies the SP 4 for t'. Then CC_m also satisfies the SP 4 for t, since $V(t)=V(t')-1$. Assume $t=t'.<bcc.i.start>$ is a trace of CC_m which satisfies the SP 4 for t'. Then CC_m also satisfies the SP 4 for t, if $V(t')\leq 0$. Assume $V(t')=1$, then the last event of the trace $t'\restriction\{bcc.i.start,bcc.i.completed,bcc.i.inoperative\}$ should be bcc.i.start. According to refinement 3.4, CC_m can send another **start** message on the channel bcc.i only if it receives a **ready** message on the channel sw.i. However, according to SP 2 this is impossible before sending a **release** message on the channel sw.i. This completes the contradiction, and therefore $V(t')\leq 0$. []

SP 5 : IC does not send a **data** message to an EU_i without receiving a **read** message.

$$IMP\,snt\left[\bigwedge_{i=1}^{n}(0\leq V(t)=[t\downarrow bic.i.read-t\downarrow bic.i.data])\right]$$

Proof : Trivial.

The next safety property of IMP is about its deadlock-freedom. Before trying to prove this property, let us give some useful definitions, theorems and lemmas about deadlock in processes, and in networks of processes.

We can combine the deadlock properties of a process P with the concept of failures

of P as given in the following definition.

Definition 3.3.1 : The process P can deadlock after the trace s if $(s, \alpha P) \in failures(P)$. The process P is deadlock-free if $\forall s \in (\alpha P)^* \bullet (s, \alpha P) \notin failures(P)$.

Similar to deadlock properties of a process P, we can relate the failures of a network of communicating processes with its deadlock properties.

Definition 3.3.2 : A network $V =< P_i | 1 \leq i \leq n >$ can deadlock after the trace s if $(s, \alpha V) \in failures(\|_{i=1}^{n} P_i)$. A network $V =< P_i | 1 \leq i \leq n >$ is deadlock-free if $\forall s \in (\alpha V)^* \bullet (s, \alpha V) \notin failures(\|_{i=1}^{n} P_i)$.

According to definition 3.3.2, when V is deadlocked there is no P_i which is ready to engage in any event outside the vocabulary of V, since such an event does not require the agreement of another process of V.

Since parallel composition is associative, representing several nodes with a single node of parallel composition does not affect the behaviour of the network but changes the communication graph. While trying the prove the deadlock-freedom of a network of processes, we should choose the easiest network topology to work with since grouping of nodes does not affect the deadlock properties. Based on this invariancy, we can prove the deadlock-freedom of *busy* networks which are built of deadlock-free node processes.

Definition 3.3.3 : A network $V =< P_i | 1 \leq i \leq n >$ is busy if all of its node processes are deadlock-free, i.e. $\forall i \ 1 \leq i \leq n \ \forall s \in (\alpha P_i)^* \bullet (s, \alpha P_i) \notin failures(P_i)$.

In order to prove a network is busy, we need to show every node process is deadlock-

free. When node processes conform to a simple subset of CSP, it is easy to prove that the network is busy. The following rule is given in [Ros 87], to prove the deadlock-freedom of processes constructed of a subset of CSP. In lemma 3.5, SKIP denotes the successful termination of a process; (P;Q) denotes the sequential composition of processes P then Q; and (μp.P) denotes the recursion.

Lemma 3.5 : Assume the definition of the process P is constructed from the following syntax :

$P ::= SKIP | a \rightarrow P | P; Q | P \,[]\, Q | P \sqcap Q | f(P) | p | \mu p.P$

where p denotes a process variable and P is a divergence-free process which does not have any free process variables and every occurrence of SKIP in P is followed by a sequential composition operator. Then P is deadlock-free.

The following theorem is given in [Bro 91] to prove the deadlock-freedom of busy networks.

Theorem 3.3 : Let $V =< P_i | 1 \leq i \leq n >$ be a busy network with vocabulary Λ. If V is free of strong Λ–conflict, any deadlock state of the network contains a proper cycle of ungranted requests with respect to Λ. If V is conflict-free then any deadlock state contains a proper cycle of at least three ungranted requests with respect to Λ, such that the only requests being made in this state between processes involved in the cycle are the requests recorded in the cycle.

Proof : [Bro 91].

According to theorem 3.3, in order to show a network V is deadlock-free, we first have to show that it is free of Λ–conflict and then that there may not be any proper cycle of ungranted requests with respect to Λ.

In order to show that a network V is free of strong αIMP-conflict, we need to show all pairs of processes are free of Λ–conflict according to definition 3.1.10. Lemmas 3.6–3.8 [Bro 91] are given to be used to prove conflict-freedom of pairs of processes, however we will skip the proofs of these lemmas.

In each of the following theorems, it is assumed that $(\alpha P \cap \alpha Q) \subseteq \Gamma$ and P and Q are deadlock-free.

Lemma 3.6 : The pair <P,Q> is free of Γ–conflict whenever $|\alpha P \cap \alpha Q| \leq 1$.

Lemma 3.7 : The pair <P,Q> is free of Γ–conflict if there is an infinite sequence of events common to the alphabets of P and Q, say $u \in (\alpha P \cap \alpha Q)^{w}$, such that in every trace of P and in every trace of Q the communications between P and Q form a prefix of u.

Lemma 3.8 : The pair <P,Q> is free of Γ–conflict if for every trace s of P||Q, whenever $(s \uparrow \alpha P, X) \in failures(P)$ and $(s \uparrow \alpha Q, Y) \in failures(Q)$ satisfy $X \supseteq \alpha P - \Gamma, Y \supseteq \alpha Q - \Gamma, (\alpha P - X) \cap \alpha Q \neq \emptyset$, and $(\alpha Q - Y) \cap \alpha P \neq \emptyset$, it follows that either $X \cap initials(Q\ after\ s \uparrow \alpha Q) = \emptyset$ or $Y \cap initials(P\ after\ s \uparrow \alpha P) = \emptyset$.

Now we can apply theorem 3.3 to prove that IMP is deadlock-free.

SP 6 : IMP is deadlock-free.

Proof : According to refinement 3.4 every process obeys the CSP subset given in lemma 3.5. Since every recursive call is preceded by a communication in all the processes of refinement 3.4, all recursions are guarded and therefore all the processes are divergence-free. Consequentially, the network IMP is busy.

Since the processes which do not communicate are trivially free of Λ–conflict; in the following, all the possible pairs of communicating processes are analyzed in order to show that they are free of Λ–conflict.

i) **<CC_m,SW_i>** is free of Λ–conflict, since SW_i cannot refuse anything CC_m may offer (Lemma 3.8).

ii) $\forall i.\ 1 \leq i < n$ **<SW_i,SW_{i+1}>** is free of Λ–conflict, since SW_{i+1} cannot refuse anything SW_i may offer (Lemma 3.8).

iii) **<CC_m,QUE>** is free of Λ–conflict, since QUE cannot refuse anything CC_m may offer (Lemma 3.8).

iv) **<CC_m,$BUFCC_n$>** is free of Λ–conflict, since $|\alpha CC_m \cap \alpha BUFCC_n| = 1$ (Lemma 3.6).

v) $\forall i.\ 1 \leq i < n$ **<$BUFCC_i$,$BUFCC_{i+1}$>** is free of Λ–conflict, since $|\alpha BUFCC_i \cap \alpha BUFCC_{i+1}| = 1$ (Lemma 3.6).

vi) $\forall i.\ 1 \leq i \leq n$ **<EU_i,$BUFCC_1$>** is free of Λ–conflict, since $|\alpha EU_i \cap \alpha BUFCC_1| = 1$ (Lemma 3.6).

vii) $\forall i.\ 1 \leq i \leq n$ **<CC_m,BCC_i>** is free of Λ–conflict, since CC_m cannot refuse anything BCC_i may offer (Lemma 3.8).

viii) $\forall i.\ 1 \leq i \leq n$ **<EU_i,BCC_i>** is free of Λ–conflict, since EU_i cannot refuse anything BCC_i may offer (Lemma 3.8).

ix) $\forall i.\ 1 \leq i \leq n$ **<EU_i,BIC_i>** is free of Λ–conflict, since EU_i cannot refuse anything BIC_i may offer (Lemma 3.8).

x) $\forall i.\ 1 \leq i \leq n$ **<EU_i,$BUFIC_1$>** is free of Λ–conflict, since $|\alpha EU_i \cap \alpha BUFIC_1| = 1$ (Lemma 3.6).

xi) **<IC,$BUFIC_n$>** is free of Λ–conflict, since $|\alpha IC \cap \alpha BUFIC_n| = 1$ (Lemma 3.6).

xii) $\forall i.\ 1 \leq i \leq n$ **<IC,BIC$_i$>** is free of Λ–conflict, since IC cannot refuse anything BIC_i may offer (Lemma 3.8).

xiii) $\forall i.\ 1 \leq i < n$ **<BUFIC$_i$,BUFIC$_{i+1}$>** is free of Λ–conflict, since $|\alpha BUFIC_i \cap \alpha BUFIC_{i+1}| = 1$ (Lemma 3.6).

Now we can prove that a proper cycle of ungranted requests cannot occur in IMP. In the following, all the possible cycles according to the communication graph are analyzed.

i) **<(CC$_m$,SW$_i$),(SW$_i$,SW$_{i+1}$),...,(SW$_j$,CC$_m$)>** : There may not be an ungranted request from SW_i to SW_{i+1} unless there are consecutive **get**, **check**, or **diag**nostic messages in the trace of CC_m. However, according to SP 1, CC_m does not send another **get**, **check**, or **diag**nostic message before receiving a reply for the previous one.

ii) **<(CC$_m$,BCC$_i$),(BCC$_i$,EU$_i$),(EU$_i$,BUFCC$_1$),...,(BUFCC$_n$,CC$_m$)>** : There may not be a request from BCC_i to EU_i when there is a request from EU_i to $BUFCC_1$ unless CC_m sends a **start** message to an EU_i which is already active. However, according to SP 4, this cannot happen.

iii) **<(IC,BIC$_i$),(BIC$_i$,EU$_i$),(EU$_i$,BUFIC$_1$),...,(BUFIC$_n$,IC)>** : There may not be a request from BIC_i to EU_i when there is a request from EU_i to $BUFIC_1$ unless IC sends a **data** message to an EU_i without receiving a **read** message. However, according to SP 5, this cannot happen.

iv) **<(EU$_i$,BUFCC$_1$),...,(BUFCC$_n$,CC$_m$),(CC$_m$,BCC$_j$),(BCC$_j$,EU$_j$),(EU$_j$,BIC$_j$),(BIC$_j$,IC),(IC,BIC$_i$),(BIC$_i$,EU$_i$)>** : There may not be a request from BCC_j to EU_j when there is a request from EU_j to BIC_j unless CC sends a **start** message to an EU_j which is already active. However, according to SP 4, this cannot happen.

v) **<(EU$_i$,BUFCC$_1$),...,(BUFCC$_n$,CC$_m$),(CC$_m$,BCC$_j$),(BCC$_j$,EU$_j$),(EU$_j$,BUFIC$_1$)**

,...,(BUFIC$_n$,IC),(IC,BIC$_i$),(BIC$_i$,EU$_i$)> : There may not be a request from BCC$_j$ to EU$_j$ when there is a request from EU$_j$ to BUFIC$_1$ unless CC sends a **start** message to an EU$_j$ which is already active. However, according to SP 4, this cannot happen.

vi) **<(EU$_i$,BCC$_i$),(BCC$_i$,CC$_m$),(CC$_m$,BCC$_j$),(BCC$_j$,EU$_j$),(EU$_j$,BIC$_j$),(BIC$_j$,IC), (IC,BIC$_i$),(BIC$_i$,EU$_i$)>** : Same as (iv).

vii) **<(EU$_i$,BCC$_i$),(BCC$_i$,CC$_m$),(CC$_m$,BCC$_j$),(BCC$_j$,EU$_j$),(EU$_j$,BUFIC$_1$),..., (BUFIC$_n$,IC),(IC,BIC$_i$),(BIC$_i$,EU$_i$)>** : Same as (v).

Therefore, a proper cycle of ungranted requests with respect to Λ cannot occur. This completes the proof of SP 6. []

SP 7 : An active implementation can not be started.

$IMP \text{ snt } [0 \leq t\downarrow ext.req - (t\downarrow ext.res + t\downarrow ext.broken) \leq 1]$

Proof : It can be shown that if $CC_{m,conc}=CC_m \backslash (\alpha CC - \{ext.req, ext.res, ext.broken\})$ is the concealed form of the main process of the central controller, then $CC_{m,conc}=(ext.req \rightarrow (ext.res \rightarrow CC_{m,conc}) [] (ext.broken \rightarrow U_{CC,conc}))$ where $U_{CC,conc}=U_{CC}$.

Now we can show that the concealed form of CC$_m$, in turn IMP satisfies SP 7.

1. Assume $t\downarrow ext.req=0$, then t=<> and $V(t)=0$.

2. Assume for $t'\downarrow ext.req=n$, $0 \leq V(t') \leq 1$ holds. If $t=t'.<ext.req>$, then $V(t) \geq 0$ holds trivially since $V(t)=V(t')+1$. $V(t) \leq 1$ does not hold only if $V(t')=1$. If $V(t')=1$, the last event of the t' must be ext.req and according to the definition of CC$_{m,conc}$ this is a contradiction. Similarly, if $t=t'.<ext.res> \vee t=t'.<ext.broken>$, then $V(t) \leq 1$ holds trivially since $V(t)=V(t')-1$. $V(t) \geq 0$ does not hold only if $V(t')=0$. If $V(t')=0$, the last event of the t' must be either ext.res or ext.broken and according to the definition of CC$_{m,conc}$ this is a contradiction. []

SP 8 : An inoperative implementation can not give a response.

IMP snt $[(t[\#t]{=}ext.broken \wedge s{=}t.u) \Rightarrow (u \downarrow ext.res{=}0)]$

Proof : According to refinement 3.4, CC_m snt $[(t[\#t]{=}ext.broken \wedge s{=}t.u) \Rightarrow (u \leq <ext.req, ext.broken>^*)]$.

Therefore, $u \downarrow ext.res{=}0$. []

3.3.3 Liveness of IMP

In order to prove the liveness of the implementation, we need to make certain assumptions about the *fairness* properties of the processes. These assumptions are necessary to guarantee that each process makes some progress. Fairness assumptions do not affect the safety properties of a system. They are liveness properties of the system.

Fairness Assumption 1 : A continuously enabled action must eventually be executed.

The fairness assumption 1 is not strong enough to limit the delay associated with the transfer of a message from an execution unit to the CC or IC. According to the next assumption, any message waits for delivery and computation of at most n-1 messages.

Fairness Assumption 2 : During the execution of $BUFCC_1$ ($BUFIC_1$), if there is more than one **trig**ger message coming from the execution units, then there will be no new **trig**ger messages coming from the other execution units until the sent ones are placed in the BUF_{CC} (BUF_{IC}).

The next fairness assumption is necessary to exclude the possibility of overtaking of 1–size buffers of an execution unit by either the execution unit or the controller.

Fairness Assumption 3 : $\forall i.\ 1 \leq i \leq n$ BCC_i (BIC_i) is fair to its two channels used for communicating with the EU_i and the CC (IC).

Now, using the given fairness assumptions we can prove the liveness properties of the implementation.

Liveness Property (LP) 1 : Each **read** and **write** message sent by an execution unit is eventually received by the interface controller.

$$[\bigwedge_{i=1}^{n}(((t[j]=eu.i.read)\Rightarrow(\exists k.\ k>j\wedge(t[k]=bic.i.read)))\wedge$$
$$((t[j]=eu.i.write)\Rightarrow(\exists k.\ k>j\wedge(t[k]=bic.i.write))))]$$

Proof : $[\bigwedge_{i=1}^{n}(((t[j]=eu.i.read)\vee(t[j]=eu.i.write))\Rightarrow(\exists k.\ k>j\wedge(t[k]=bufic.1.i.trig)))$ follows from the fairness assumptions 1 and 2. Based on the fairness assumption 1, $[\bigwedge_{i=1}^{n}((t[j]=bufic.1.i.trig)\Rightarrow(\exists k.\ k>j\wedge(t[k]=bufic.eun(i))))$. According to the fairness assumptions 1 and 3, $[\bigwedge_{i=1}^{n}((t[j]=bufic.eun(i))\Rightarrow(\exists k.\ k>j\wedge(t[k]=bic.i.ready)))$. Since IC is able to receive either **read** and **write** message after sending a **ready** message and following the fairness assumption 1, $[\bigwedge_{i=1}^{n}((t[j]=bic.i.ready)\Rightarrow(\exists k.\ k>j\wedge((t[k]=bic.i.read)\vee(t[k]=bic.i.write))))$. []

LP 2 : Each **new**, **completed** and **write** message sent by an execution unit is eventually received by the central controller.

$$[\bigwedge_{i=1}^{n}(((t[j]=eu.i.new)\Rightarrow(\exists k.\ k>j\wedge(t[k]=bcc.i.new)))\wedge$$
$$((t[j]=eu.i.completed)\Rightarrow(\exists k.\ k>j\wedge(t[k]=bcc.i.completed)))\wedge$$
$$((t[j]=eu.i.inoperative)\Rightarrow(\exists k.\ k>j\wedge(t[k]=bcc.i.inoperative))))]$$

Proof : Similar to LP 1.

LP 3 : An execution unit eventually receives a **data** message for each **read** message it has sent.

$$\left[\bigwedge_{i=1}^{n}((t[j]=eu.i.read)\Rightarrow(\exists k.\ k>j\wedge(t[k]=eu.i.data)))\right]$$

Proof : According to the LP 1,

$\left[\bigwedge_{i=1}^{n}((t[j]=eu.i.read)\Rightarrow(\exists k.\ k>j\wedge(t[k]=bic.i.read)))\right]$. According to the fairness assumption 1,

$\left[\bigwedge_{i=1}^{n}((t[j]=bic.i.read)\Rightarrow(\exists k.\ k>j\wedge(t[k]=bic.i.data)))\right]$ since the BIC_i is ready to receive the **data** message. Similarly,

$\left[\bigwedge_{i=1}^{n}((t[j]=bic.i.data)\Rightarrow(\exists k.\ k>j\wedge(t[k]=eu.i.data)))\right]$. []

LP 4 : An execution unit eventually sends either a **completed** or an **inoperative** message for each **start** message it has received.

$\left[\bigwedge_{i=1}^{n}((t[j]=eu.i.start)\Rightarrow(\exists k.\ k>j\wedge(t[k]=bcc.i.completed\vee t[k]=bcc.i.inoperative)))\right]$

Proof : Since the buffers of an idle execution unit are empty, based on LP 2, LP 3 and the fairness assumption 1,

$\left[\bigwedge_{i=1}^{n}((t[j]=eu.i.start)\Rightarrow(\exists k,l,m.\ m>l>k>j\wedge(t[k]=bic.i.read)\wedge(t[l]=bic.i.data)\wedge(t[m]=bic.i.write)))\right]$.

Since each node of input value is processed once according to algorithms 2.1–2.4, the number of nodes which can be reached from a starting node should be bounded. Following

$\left[\bigwedge_{i=1}^{n}(((t[j]=eu.i.read)\wedge(w(j)=w))\Rightarrow(\exists k.\ k>j\wedge(t[k]=eu.i.read)\wedge(w(k)=w-1)))\right]$, the number of remaining nodes to be processed is a monotonically decreasing function. Therefore, when there are no more nodes to be processed, an execution unit either sends a **completed** message or an **inoperative** message. According to the LP 2, this message is eventually delivered to the CC. []

LP 5 : An implementation eventually sends either a **res**ponse or a **broken** message for each **req**uest message it has received.

$[(t[i]=ext.req)\Rightarrow(\exists j.\ j>i\wedge(t[j]=ext.res\vee t[j]=ext.broken))]$

Proof : According to the refinement 3.4, it can be shown that if all the execution units

are inoperative, due to the fairness assumption, CC_m sends a **broken** message for the **req**uest message. Otherwise, CC_m finds an idle execution unit and sends a **start** message.

$[(t[i]=ext.req)\Rightarrow((t[i+1]=ext.broken)\vee(\exists j.\ j>i\wedge(t[j]=eu.k.start)))]$. According to LP 4, each execution unit eventually sends either a **completed** message or an **inoperative** message for each **start** message it has received after finishing the processing of p nodes if there are p-1 nodes reachable from the initial node. Therefore, the number of remaining nodes to be processed is a monotonically decreasing function. Based on the previous liveness properties and the fairness assumptions, if all the nodes of the input value are processed before all the execution units become inoperative, CC_m sends a **res**ponse message. Otherwise, an **inoperative** message is sent. []

3.4 Complexity

Until now, we have analyzed the properties of IMP related with its correctness. For a distributed implementation, analysis of complexity has equal importance. There are two basic complexity measures for distributed algorithms; time and message complexity. The time complexity of an algorithm measures the time needed both for computations of processes and transmission of messages. For introduced algorithms, we assume that computational and transmission costs are comparable. The total number of messages is used as the measure of message complexity.

As explained before, corresponding phases of encoding and decoding are inverses of each other; thus, their complexities are the same. Therefore, in the following we only give complexity analysis about distributed implementations of algorithms for phases of decoding.

Lemma 3.9 : For distributed implementation of algorithm 2.1, when the number of nodes in the intermediate value tree is N,

a) Message complexity is O(N).

b) Worst-case time complexity is O(N), and average time complexity is O(log N) with $O(\frac{N}{log\ N})$ EU's assuming that the average number of execution units after the transient phase is proportional to the number of execution units.

Proof : a) The number of messages is proportional to the number of individual transformations, one for each node of the intermediate value tree. In other words,

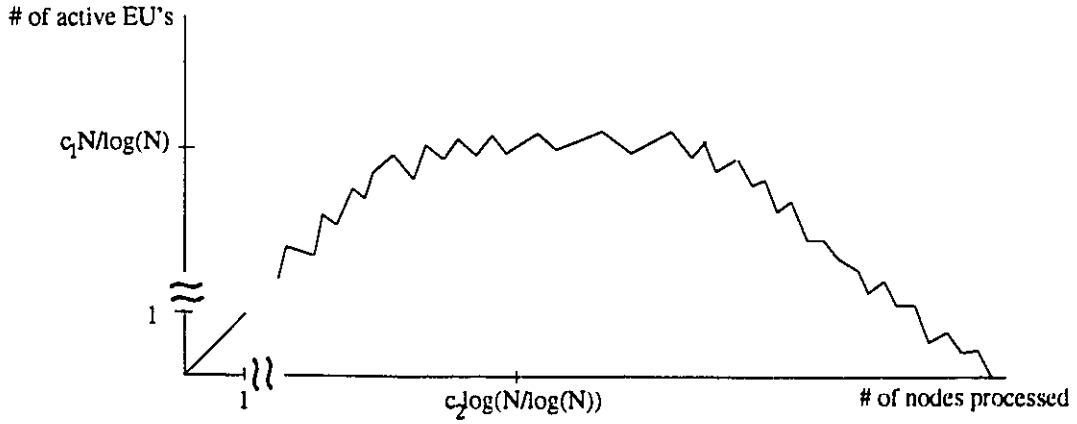
$$IMP\ sat\ \left[(t[1]=ext.req \wedge t[\#t]=ext.res) \Rightarrow \sum_{i=1}^{n}(t\downarrow eu.i.read)=\sum_{i=1}^{n}(t\downarrow eu.i.data)=\sum_{i=1}^{n}(t\downarrow eu.i.write)=N\right].$$

Therefore, the total number of messages is O(N).

b) Under the worst-case scenario, EU_1 receives the initial **start** message, and then does not issue any **new** message until it finishes the processing of entire input in O(N) time. In other words,

$$\left[(t[j]=ext.req \wedge (\exists k,l.\ l>k>j \wedge t[k]=eu.i.start \wedge t[l]=ext.res) \wedge (\forall m.\ k<m<l \wedge t[m]\neq eu.i.new)) \Rightarrow \left(\sum_{p=1_{p\neq i}}^{n}(t\downarrow eu.p.read+t\downarrow eu.p.data+t\downarrow eu.p.write)=0\right) \wedge ((t\downarrow eu.i.read)=(t\downarrow eu.i.data)=(t\downarrow eu.i.write)=N)\right]$$

Figure 3.2 Number of Active EU's During Computation



For the average case, EU's send **new** messages and more than one execution unit becomes active during the computation. The number of active execution units as a function of the number of processed nodes is given in Fig. 3.2. According to Fig. 3.2, initially only one execution unit is activated. During the course of computation the number of active execution units increases. Since $|S_{j,k}| \leq 2$, $c_1\left(\frac{N}{log\,N}\right)$ execution units may become active only after $c_2 log\left(\frac{N}{log\,N}\right)$ steps. Since the average number of active execution units after the transient phase is assumed to be proportional to the number of execution units, i.e. $c_1\left(\frac{N}{log\,N}\right)$, the time complexity becomes O(log N). []

Lemma 3.10 : For distributed implementation of algorithm 2.2, when the number of nodes in the intermediate value tree is N and the total number of nodes in the global type tree is M, where the height of both trees is k, the global type tree is well-balanced,

a) Message complexity is $O(\sqrt[k]{M}N)$.

b) Worst-case time complexity is $O(\sqrt[k]{M}N)$, and average time complexity is $O(\sqrt[k]{M}\ log\,N)$ with $O(\frac{N}{log\,N})$ EU's assuming that the average number of execution units after the transient phase is proportional to the number of execution units.

Proof : a) A given value tree node is compared with m nodes of the global type tree. Assume that each non-leaf node of the global type tree has c children. Under this assumption, we can write $\sum_{i=0}^{k} c^i = M$. Assuming that $c \geq 2$, $c^{k+1} > M > c^k$. Using the upper limit for c, it is found that $c = \sqrt[k]{M}$. According to the algorithm 2.2, in the best case m, the number of nodes of the global type tree compared with a given value node, is 1 if the matching value and type nodes are in the same order and their numbers are equal. However, we can use the average case where m is proportional to c. In this case, the total number of unit transformations, and

in turn the total number of messages become O($\sqrt[k]{MN}$).

b) Under the worst-case scenario, EU_1 receives the initial **start** message, and then does not issue any **new** message until it finishes the processing of entire input in O($\sqrt[k]{MN}$) time. For the average case, EU's send **new** messages and more than one execution unit becomes active during the computation. Assuming that the number of active execution units as a function of the number of processed nodes is similar to that of which is given for algorithm 2.1 $c_1\left(\frac{N}{log\,N}\right)$ execution units may become active only after $c_2 log\left(\frac{N}{log\,N}\right)$ steps. Based on the same assumption of the proportionality of the average number of active execution units after the transient phase to the number of execution units, i.e. $c_1\left(\frac{N}{log\,N}\right)$, the time complexity becomes O($\sqrt[k]{M}\ log\,N$). []

Chapter 4 PDU ENCODING / DECODING ARCHITECTURES

In this chapter, we will discuss different architectural models for combining implementations for different phases of encoding and decoding. The models are based on the distributed implementation model discussed in Chapter 3.

In the first section, single PDU encoder / decoder (ED) architectures are discussed. A taxonomy is given for single ED architectures based on the number and organization of IMP modules. The taxonomy introduces single EDs for encoding and decoding PDUs of a single layer coming from a single source. Multiple-layer and multiple-source single ED architectures are also defined as refinements obtained from the initial taxonomy.

The second section deals with multiple ED architectures which are composed of a number of single EDs of different architectural models. Multiple ED architectures can be used to serve a single source or a number of sources. Analysis of multiple ED architectures for multiple-sources is divided according to the mapping between the sources and EDs.

4.1 Single PDU Encoder / Decoder Architectures

A PDU ED is a single IMP or a composition of IMPs which are collectively capable of providing responses for both PDU encoding and decoding requests. Since the distributed implementation model IMP is unique for all phase transformations, the same IMP can be used to perform different transformations provided that an external request message sent to CC of an IMP specifies the requested transformation. In all

definitions given in this chapter, $\mathcal{P}Rq, \mathcal{M}_dRq, \mathcal{M}_eRq, \mathcal{A}Rq$ are used to denote requests for parsing, type-value matching for decoding, type-value matching for encoding, and assembling transformations, respectively. Similarly, $\mathcal{P}Re, \mathcal{M}_dRe, \mathcal{M}_eRe, \mathcal{A}Re$ are used to denote responses for parsing, type-value matching for decoding, type-value matching for encoding, and assembling transformations, respectively.

4.1.1 Taxonomy

Single EDs serving encoding and decoding requests for a single layer coming from a single source are divided according to the number and configuration of IMPs. There are four different models for single-layer single-source single EDs.

4.1.1.1 (PM_dM_eA) Model

If a single IMP is to provide all the phase transformations for encoding and decoding requests, the (PM_dM_eA) model is used. The (PM_dM_eA) model is the composition of an interface component, S-IPMMA, and the single IMP renamed as S-(PM_dM_eA). S-IPMMA links S-(PM_dM_eA) to the environment and controls the order of phases of encoding and decoding (Fig. 4.1).

In the IMP renamed as S-(PM_dM_eA), the CC accepts $\mathcal{P}Rq, \mathcal{M}_dRq, \mathcal{M}_eRq, \mathcal{A}Rq$ at the gate pmma (new name of ext) and responds with $\mathcal{P}Re, \mathcal{M}_dRe, \mathcal{M}_eRe, \mathcal{A}Re$. In all the models of this chapter, details related with IMP are hidden.

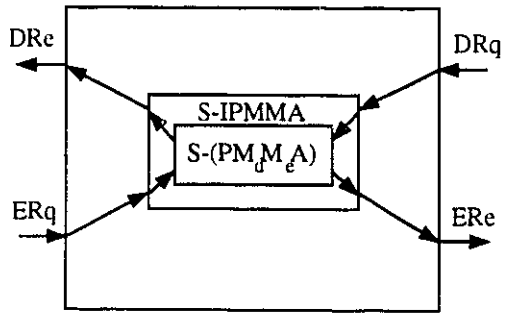
(PM_dM_eA) Description

$$(PM_dM_eA)=S\text{-}IPMMA \| S\text{-}(PM_dM_eA)$$

$$S\text{-}IPMMA=(ext?DRq \rightarrow pmma!\mathcal{P}Rq \rightarrow pmma?\mathcal{P}Re \rightarrow pmma!\mathcal{M}_dRq \rightarrow pmma?\mathcal{M}_dRe \rightarrow ext!DRe \rightarrow S\text{-}IPMMA)\,[]$$
$$(ext?ERq \rightarrow pmma!\mathcal{M}_eRq \rightarrow pmma?\mathcal{M}_eRe \rightarrow pmma!\mathcal{A}Rq \rightarrow pmma?\mathcal{A}Re \rightarrow ext!ERe \rightarrow S\text{-}IPMMA)$$

$$S\text{-}(PM_dM_eA)=(pmma?\mathcal{P}Rq \rightarrow pmma!\mathcal{P}Re \rightarrow S\text{-}(PM_dM_eA))\,[]$$
$$(pmma?\mathcal{M}_dRq \rightarrow pmma!\mathcal{M}_dRe \rightarrow S\text{-}(PM_dM_eA))\,[]$$
$$(pmma?\mathcal{M}_eRq \rightarrow pmma!\mathcal{M}_eRe \rightarrow S\text{-}(PM_dM_eA))\,[]$$
$$(pmma?\mathcal{A}Rq \rightarrow pmma!\mathcal{A}Re \rightarrow S\text{-}(PM_dM_eA))$$

Figure 4.1 Single-Layer Single-Source (PM_dM_eA) Model



4.1.1.2 (PM_d)-(M_eA) Model

In chapter 2, it is shown that both encoding and decoding can be divided into two phase transformations. According to the (PM_d)-(M_eA) model, an IMP is used for phases of decoding, whereas another IMP is used for phases of encoding. The (PM_d)-(M_eA) model is the composition of two interface components, S-IPM and S-IMA with two IMPs

renamed as S-(PM_d) and S-(M_eA). S-IPM and S-IMA link S-(PM_d) and S-(M_eA) to the environment and control the phases of decoding and encoding, respectively (Fig. 4.2).

(PM_d)-(M_eA) Description

$$(PM_d)\text{-}(M_eA)=S\text{-}IPM \| S\text{-}(PM_d) \| S\text{-}IMA \| S\text{-}(M_eA)$$

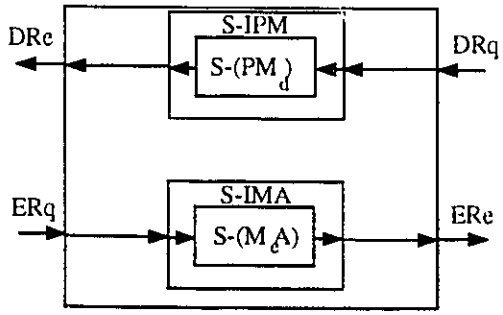
$$S\text{-}IPM=(ext?DRq \rightarrow pm!\mathcal{P}Rq \rightarrow pm?\mathcal{P}Re \rightarrow pm!\mathcal{M}_dRq \rightarrow pm?\mathcal{M}_dRe \rightarrow ext!DRe \rightarrow S\text{-}IPM)$$

$$S\text{-}(PM_d)=(pm?\mathcal{P}Rq \rightarrow pm!\mathcal{P}Re \rightarrow S\text{-}(PM_d)) \square (pm?\mathcal{M}_dRq \rightarrow pm!\mathcal{M}_dRe \rightarrow S\text{-}(PM_d))$$

$$S\text{-}IMA=(ext?ERq \rightarrow ma!\mathcal{M}_eRq \rightarrow ma?\mathcal{M}_eRe \rightarrow ma!\mathcal{A}Rq \rightarrow ma?\mathcal{A}Re \rightarrow ext!ERe \rightarrow S\text{-}IMA)$$

$$S\text{-}(M_eA)=(ma?\mathcal{M}_eRq \rightarrow ma!\mathcal{M}_eRe \rightarrow S\text{-}(M_eA)) \square (ma?\mathcal{A}Rq \rightarrow ma!\mathcal{A}Re \rightarrow S\text{-}(M_eA))$$

Figure 4.2 Single-Layer Single-Source (PM_d)-(M_eA) Model



4.1.1.3 (PA)-(M_dM_e) Model

Another single ED model with two IMPs is the (PA)-(M_dM_e) model. According to the (PA)-(M_dM_e) model, complementary phases of encoding and decoding are performed in the same IMP. The (PA)-(M_dM_e) model is the composition of a unique interface component S-I(PA)-(MM) with two IMPs renamed as S-(PA) and S-(M_dM_e). The interface component is unique to make the resulting ED deadlock-free. Since, encoding and decoding requests use both IMPs, there is a deadlock possibility due to conflict if a separate interface is used for each IMP. The two-tuple index of S-I(PA)-(MM) respectively

shows the states of S-(PA) and S-(M_dM_e) as either idle, or used for encoding or decoding (Fig. 4.3).

(PA)-(M_dM_e) Description

$$(PA)-(M_dM_e)=S-(PA)\|(S-I(PA)-(MM))_{<i,i>}\|S-(M_dM_e)$$

$$S-(PA)=(pa?\mathcal{P}Rq\rightarrow pa!\mathcal{P}Re\rightarrow S-(PA))\square(pa?\mathcal{A}Rq\rightarrow pa!\mathcal{A}Re\rightarrow S-(PA))$$

$$(S-I(PA)-(MM))_{<i,i>}=(ext?DRq\rightarrow pa!\mathcal{P}Rq\rightarrow(S-I(PA)-(MM))_{<d,i>})\square$$
$$(ext?ERq\rightarrow mm!\mathcal{M}_eRq\rightarrow(S-I(PA)-(MM))_{<i,e>})$$

$$(S-I(PA)-(MM))_{<d,i>}=(ext?ERq\rightarrow mm!\mathcal{M}_eRq\rightarrow(S-I(PA)-(MM))_{<d,e>})\square$$
$$(pa?\mathcal{P}Re\rightarrow mm!\mathcal{M}_dRq\rightarrow(S-I(PA)-(MM))_{<i,d>})$$

$$(S-I(PA)-(MM))_{<i,e>}=(ext?DRq\rightarrow pa!\mathcal{P}Rq\rightarrow(S-I(PA)-(MM))_{<d,e>})\square$$
$$(mm?\mathcal{M}_eRe\rightarrow pa!\mathcal{A}Rq\rightarrow(S-I(PA)-(MM))_{<e,i>})$$

$$(S-I(PA)-(MM))_{<d,e>}=(mm?\mathcal{M}_eRe\rightarrow pa?\mathcal{P}Re\rightarrow pa!\mathcal{A}Rq\rightarrow mm!\mathcal{M}_dRq\rightarrow(S-I(PA)-(MM))_{<e,d>})\square$$
$$(pa?\mathcal{P}Re\rightarrow mm?\mathcal{M}_eRe\rightarrow mm!\mathcal{M}_dRq\rightarrow pa!\mathcal{A}Rq\rightarrow(S-I(PA)-(MM))_{<e,d>})$$

$$(S-I(PA)-(MM))_{<i,d>}=(ext?DRq\rightarrow pa!\mathcal{P}Rq\rightarrow(S-I(PA)-(MM))_{<d,d>})\square$$
$$(mm?\mathcal{M}_dRe\rightarrow ext!DRe\rightarrow(S-I(PA)-(MM))_{<i,i>})$$

$$(S-I(PA)-(MM))_{<e,i>}=(ext?ERq\rightarrow mm!\mathcal{M}_eRq\rightarrow(S-I(PA)-(MM))_{<e,e>})\square$$
$$(pa?\mathcal{A}Re\rightarrow ext!ERe\rightarrow(S-I(PA)-(MM))_{<i,i>})$$

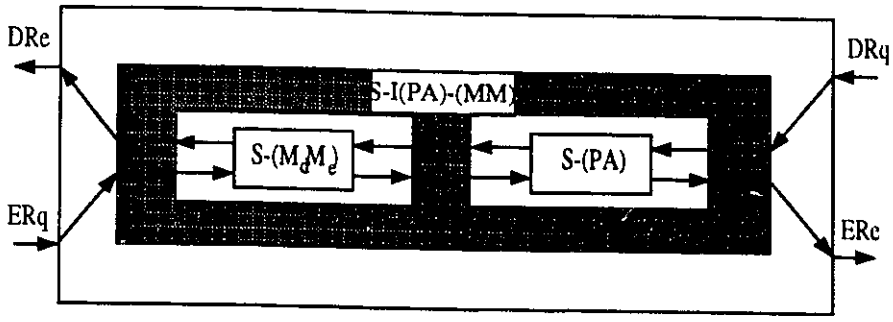
$$(S-I(PA)-(MM))_{<e,d>}=(mm?\mathcal{M}_dRe\rightarrow ext!DRe\rightarrow(S-I(PA)-(MM))_{<e,i>})\square$$
$$(pa?\mathcal{A}Re\rightarrow ext!ERe\rightarrow(S-I(PA)-(MM))_{<i,d>})$$

$$(S-I(PA)-(MM))_{<d,d>}=(mm?\mathcal{M}_dRe\rightarrow ext!DRe\rightarrow(S-I(PA)-(MM))_{<d,i>})$$

$$(S-I(PA)-(MM))_{<e,e>}=(pa?\mathcal{A}Re\rightarrow ext!ERe\rightarrow(S-I(PA)-(MM))_{<i,e>})$$

$$S-(M_dM_e)=(mm?\mathcal{M}_dRq\rightarrow mm!\mathcal{M}_dRe\rightarrow S-(M_dM_e))\square(mm?\mathcal{M}_eRq\rightarrow mm!\mathcal{M}_eRe\rightarrow S-(M_dM_e))$$

Figure 4.3 Single-Layer Single-Source (PA)-(M_dM_e) Model



4.1.1.4 (P)-(M_d)-(M_e)-(A) Model

According to the (P)-(M_d)-(M_e)-(A) model, each phase of encoding and decoding is performed in a separate IMP. The (P)-(M_d)-(M_e)-(A) model is the composition of four interface components, S-IP, S-IM_d, S-IM_e, and S-IA with four IMPs renamed as S-(P), S-(M_d), S-(M_e), and S-(A). Interconnected S-IP and S-IM_d link S-(P) and S-(M_d) to the environment for decoding requests and responses, respectively. Similarly, interconnected S-IM_e and S-IA link S-(M_e) and S-(A) to the environment for encoding requests and responses, respectively (Fig. 4.4).

(P)-(M_d)-(M_e)-(A) Description

$$(P)-(M_d)-(M_e)-(A)=S-IP\|S-(P)\|S-IM_d\|S-(M_d)\|S-IM_e\|S-(M_e)\|S-IA\|S-(A)$$

$$S-IP=(ext?DRq\rightarrow p!\mathcal{P}Rq\rightarrow p?\mathcal{P}Re\rightarrow im_d!DRq\rightarrow S-IP)$$

$$S-(P)=(p?\mathcal{P}Rq\rightarrow p!\mathcal{P}Re\rightarrow S-(P))$$

$$S-IM_d=(im_d?DRq\rightarrow m_d!\mathcal{M}_dRq\rightarrow m_d?\mathcal{M}_dRe\rightarrow ext!DRe\rightarrow S-IM_d)$$

$$S-(M_d)=(m_d?\mathcal{M}_dRq\rightarrow m_d!\mathcal{M}_dRe\rightarrow S-(M_d))$$

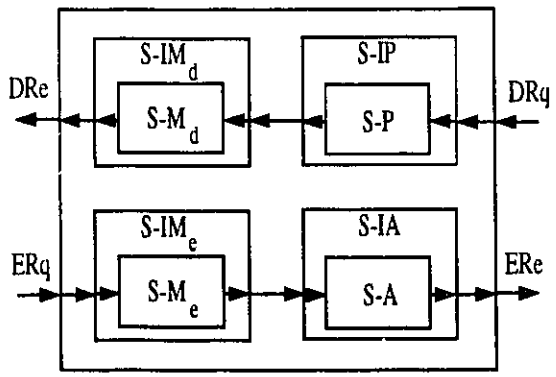
$$S-IM_e=(ext?ERq\rightarrow m_e!\mathcal{M}_eRq\rightarrow m_e?\mathcal{M}_eRe\rightarrow ia!ERq\rightarrow S-IM_e)$$

$$S-(M_e)=(m_e?\mathcal{M}_eRq\rightarrow m_e!\mathcal{M}_eRe\rightarrow S-(M_e))$$

$$S-IA=(ia?ERq\rightarrow a!\mathcal{A}Rq\rightarrow a?\mathcal{A}Re\rightarrow ext!ERe\rightarrow S-IA)$$

$$S-(A)=(a?\mathcal{A}Rq\rightarrow a!\mathcal{A}Re\rightarrow S-(A))$$

Figure 4.4 Single-Layer Single-Source (P)-(M_d)-(M_e)-(A) Model



4.1.2 Multiple-Layer Single EDs

The set of layers served by an ED_i is $S_l(ED_i)$ called the *layer set* of ED_i. If $|S_l(ED_i)| = 1$, e.g. if ED_i serves a unique layer, then each IMP of ED_i performs only the associated phase transformations defined for the layer L_j where $S_l(ED_i) = \{L_j\}$.

If $|S_l(ED_i)| > 1$, e.g. if ED_i serves a number of layers, then each IMP of ED_i should be able to perform all the associated phase transformations defined for the layers $L_1,...,L_L$ where $S_l(ED_i) = \{L_1, ..., L_L\}$. If an ED serves multiple layers, then each encoding and decoding request should be accompanied by a layer identifier which specifies the layer whose PDU is to be encoded or decoded.

As an example of multiple-layer single EDs, $(PM_dM_eA)_{[1,...,L]}$ for serving encoding and decoding requests coming from L different layers is given below. $(PM_dM_eA)_{[1,...,L]}$ is obtained from the renaming of the original (PM_dM_eA) such that each request and response is indexed according to its layer. $(PM_dM_eA)_{[1,...,L]}$ is shown in Fig. 4.5.

Multiple-Layer Model for (PM_dM_eA)

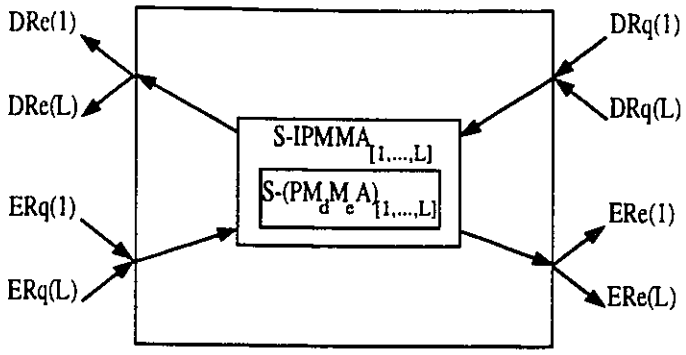
$$(PM_dM_eA)_{[1,...,L]} = S\text{-}IPMMA_{[1,...,L]} \| S\text{-}(PM_dM_eA)_{\{1,...,L\}}$$

$$S\text{-}IPMMA_{[1,...,L]} = (\Box_{i=1}^{L}(ext?DRq(i) \rightarrow pmma!\mathcal{P}Rq(i) \rightarrow pmma?\mathcal{P}Re(i) \rightarrow pmma!\mathcal{M}_dRq(i) \rightarrow pmma?\mathcal{M}_dRe(i) \rightarrow ext!DRe(i) \rightarrow S\text{-}IPMMA_{[1,...,L]})) \Box (\Box_{i=1}^{L}(ext?ERq(i) \rightarrow pmma!\mathcal{M}_eRq(i) \rightarrow pmma?\mathcal{M}_eRe(i) \rightarrow pmma!\mathcal{A}Rq(i) \rightarrow pmma?\mathcal{A}Re(i) \rightarrow ext!ERe(i) \rightarrow S\text{-}IPMMA_{[1,...,L]}))$$

$$S\text{-}(PM_dM_eA)_{[1,...,L]} = (\Box_{i=1}^{L}(pmma?\mathcal{P}Rq(i) \rightarrow pmma!\mathcal{P}Re(i) \rightarrow S\text{-}(PM_dM_eA)_{[1,...,L]})) \Box (\Box_{i=1}^{L}(pmma?\mathcal{M}_dRq(i) \rightarrow pmma!\mathcal{M}_dRe(i) \rightarrow S\text{-}(PM_dM_eA)_{[1,...,L]})) \Box (\Box_{i=1}^{L}(pmma?\mathcal{M}_eRq(i) \rightarrow pmma!\mathcal{M}_eRe(i) \rightarrow S\text{-}(PM_dM_eA)_{[1,...,L]})) \Box (\Box_{i=1}^{L}(pmma?\mathcal{A}Rq(i) \rightarrow pmma!\mathcal{A}Re(i) \rightarrow S\text{-}(PM_dM_eA)_{[1,...,L]}))$$

Since, $(PM_dM_eA)_{[1,...,L]}$ includes external events indexed according to the related layer, it cannot be regarded as a simple hiding refinement of (PM_dM_eA) without making fairness assumptions. However, in the following lemma, we can show that $(PM_dM_eA)_{[1,...,L]}$ simulates (PM_dM_eA) when its environment simulates that of (PM_dM_eA).

Figure 4.5 Multiple-Layer Single-Source (PM_dM_eA) Model



Lemma 4.1 : $f^{-1}((PM_dM_eA)_{[1,\ldots,L]}\|ENV_1)\sqsupseteq((PM_dM_eA)\|ENV)$ where

$$ENV=(ext!DRq\rightarrow ext?DRe\rightarrow ENV)\sqcap(ext!ERq\rightarrow ext?ERe\rightarrow ENV)$$

$$ENV_1=(ext!DRq(1)\rightarrow ext?DRe(1)\rightarrow ENV_1)\sqcap(ext!ERq(1)\rightarrow ext?ERe(1)\rightarrow ENV_1)$$

and

$$f^{-1}(a)=\begin{cases} b & a=b(1) \\ a & otherwise \end{cases}.$$

Proof : It can be shown that $(PM_dM_eA)_{[1,\ldots,L]}=\sqcap_{i=1}^{L} f_i(PM_dM_eA)$. Consequently, $(PM_dM_eA)_{[1,\ldots,L]}\|ENV_1=f_1(PM_dM_eA)$, and $f^{-1}((PM_dM_eA)_{[1,\ldots,L]}\|ENV_1)=((PM_dM_eA)\|ENV)$. Since $P\sqsupseteq P$, $f^{-1}((PM_dM_eA)_{[1,\ldots,L]}\|ENV_1)\sqsupseteq((PM_dM_eA)\|ENV)$. []

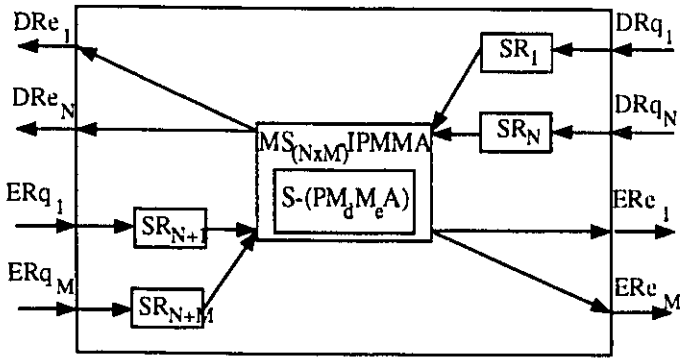
4.1.3 Multiple-Source Single EDs

The set of PDU encoding and decoding request sources served by an ED_i is $S_{ers}(ED_i)$ and $S_{drs}(ED_i)$, respectively. $S_{ers}(ED_i)$ and $S_{drs}(ED_i)$ are called the *encoding-request-source set* and *decoding-request-source set* of ED_i, respectively.

If $|S_{ers}(ED_i)| = |S_{drs}(ED_i)| = 1$, e.g. if ED_i is said to be *dedicated* to a single encoding and a single decoding request source. If $|S_{ers}(ED_i)| > 1$ ($|S_{drs}(ED_i)| > 1$), e.g. if ED_i serves a number of encoding (decoding) request sources, then ED_i is said to be *shared* among different encoding (decoding) request sources.

As an example of multiple-source single EDs, $MS_{(NxM)}$-(PM_dM_eA) for serving encoding requests coming from M sources and decoding requests coming from N sources is given below. According to the model, S-(PM_dM_eA) serves decoding and encoding requests coming from N+M sources in a cyclic fashion. Decoding requests are stored in N decoding source registers, SR_1,...,SR_N. Similarly, encoding requests are stored in M encoding source registers, SR_{N+1},...,SR_{N+M} (Fig. 4.6). Index of MS-IPMMA shows which source register the interface process is checking. Initially, MS-IPMMA checks SR_1 to see whether there is any decoding request from source 1.

Figure 4.6 Single-Layer Multiple-Source (NxM) (PM_dM_eA) Model



Multiple-Source (NxM) Model for (PM_dM_eA)

$$MS_{(N\times M)}\text{-}(PM_dM_eA)=MS_{(N\times M)}\text{-}IPMMA_1 \| S\text{-}(PM_dM_eA) \| (\|_{i=1}^{N+M} SR_{i,<>})$$

$$MS_{(N\times M)}\text{-}IPMMA_i=(sr.i!check\rightarrow(sr.i?empty\rightarrow MS_{(N\times M)}\text{-}IPMMA_{i+1})\square \qquad 1\le i\le N$$

$$(sr.i?DRq\rightarrow pmma!\mathcal{P}Rq\rightarrow pmma?\mathcal{P}Re\rightarrow pmma!\mathcal{M}_dRq\rightarrow$$

$$pmma?\mathcal{M}_dRe\rightarrow ext!DRe_i\rightarrow MS_{(N\times M)}\text{-}IPMMA_{i+1}))$$

$$MS_{(N\times M)}\text{-}IPMMA_i=(sr.i!check\rightarrow(sr.i?empty\rightarrow MS_{(N\times M)}\text{-}IPMMA_{i+1})\square \qquad N+1\le i<N+M$$

$$(sr.i?ERq\rightarrow pmma!\mathcal{M}_eRq\rightarrow pmma?\mathcal{M}_eRe\rightarrow pmma!\mathcal{A}Rq\rightarrow$$

$$pmma?\mathcal{A}Rc\rightarrow ext!ERe_{i-N}\rightarrow MS_{(N\times M)}\text{-}IPMMA_{i+1}))$$

$$MS_{(N\times M)}\text{-}IPMMA_{N+M}=(sr.N+M!check\rightarrow(sr.N+M?empty\rightarrow MS_{(N\times M)}\text{-}IPMMA_1)\square$$

$$(sr.N+M?ERq\rightarrow pmma!\mathcal{M}_eRq\rightarrow pmma?\mathcal{M}_eRe\rightarrow pmma!\mathcal{A}Rq\rightarrow$$

$$pmma?\mathcal{A}Re\rightarrow ext!ERe_M\rightarrow MS_{(N\times M)}\text{-}IPMMA_1))$$

$$SR_{i,<>}=(sr.i?check\rightarrow sr.i!empty\rightarrow SR_{i,<>})\square(ext.i?DRq\rightarrow SR_{i,<DRq>})\square \qquad 1\le i\le N+M$$

$$(ext.i?ERq\rightarrow SR_{i,<ERq>})$$

$$SR_{i,<x>}=(sr.i?check\rightarrow sr.i!x\rightarrow SR_{i,<>}) \qquad 1\le i\le N+M$$

Similar to $(PM_dM_eA)_{[1,\ldots,L]}$ $MS_{(NxM)}$-(PM_dM_eA) includes external events indexed according to the related source. Therefore, it cannot be regarded as a simple hiding refinement of (PM_dM_eA) without making fairness assumptions. However, as in Lemma 4.1, we can show that $MS_{(NxM)}$-(PM_dM_eA) simulates (PM_dM_eA) when its environment simulates that of (PM_dM_eA).

Lemma 4.2 : $g^{-1}(MS_{(N\times M)}-IPMMA_1\|(\|_{i=1}^{N+M}SR_{i,<>})\|ENV_2)\backslash S_1 \sqsupseteq (S-IPMMA\|ENV)$

where

$$S_1=\alpha g^{-1}(MS_{(N\times M)}-IPMMA_1\|(\|_{i=1}^{N+M}SR_{i,<>})\|ENV_2)-\alpha(S-IPMMA\|ENV),$$

$$ENV=(ext!DRq\rightarrow ext?DRe\rightarrow ENV)\sqcap(ext!ERq\rightarrow ext?ERe\rightarrow ENV)$$

$$ENV_2=(ext.1!DRq\rightarrow ext?DRe_1\rightarrow ENV_2)\sqcap(ext.N+1!ERq\rightarrow ext?ERe_1\rightarrow ENV_2)$$

and

$$g^{-1}(a)=\begin{cases} ext.b & a=ext.1.DRq \vee a=ext.N+1.ERq \\ ext.b & a=ext.b_1 \wedge (b=DRe \vee b=ERe) \\ a & otherwise \end{cases} .$$

Proof : It can be shown that

$$g^{-1}(MS_{(N\times M)}-IPMMA_1\|(\|_{i=1}^{N+M}SR_{i,<>})\|ENV_2)=$$

$$g^{-1}(MS_{(N\times M)}-IPMMA_1\|SR_{1,<>}\|ENV_2\|SR_{N+1,<>})$$

and $g^{-1}(MS_{(N\times M)}-IPMMA_1\|SR_{1,<>}\|ENV_2\|SR_{N+1,<>})\backslash S_1=(S-IPMMA\|ENV)$. Since $P \sqsupseteq P$, $g^{-1}(MS_{(N\times M)}-IPMMA_1\|(\|_{i=1}^{N+M}SR_{i,<>})\|ENV_2)\backslash S_1 \sqsupseteq (S-IPMMA\|ENV)$. []

4.2 Multiple PDU Encoder / Decoder Architectures

Multiple ED architectures are composed of more than one PDU EDs based on different architectural models discussed in Section 4.1.1. Multiple ED architectures are either *homogeneous* in which all the EDs are of the same single ED model, or *heterogeneous* which consists of different models of EDs. In this section, only homogeneous multiple ED architectures are discussed to limit the complexity of descriptions. However, heterogeneous ED models can easily be described as extensions of homogeneous architectures.

In this section, multiple ED architectures are classified according to the number of encoding and decoding request sources they serve. Similar to single ED architectures, multiple ED architectures may also serve a single layer or a number of layers. However, for the analysis of multiple ED architectures, the number of layers is not used as a classification factor. Instead, based on the homogeneous nature of multiple ED

architectures, multiple-layer multiple ED architectures are viewed as compositions of a number of multiple-layer single ED architectures.

4.2.1 Single-Source Multiple EDs

According to descriptions of different models of single ED architectures discussed in Section 4.1.1, an interface component becomes ready to serve an encoding or decoding request source only when it finishes serving a previous request. If the arrival rate of requests coming from a single source is smaller than the service rate of the slowest IMP used in the service, then the single ED architecture is sufficient to provide the encoding and decoding service. On the other hand, if the arrival rate of requests is much higher than the service rate of the slowest IMP used in the service, then a number of IMPs are needed to obtain a stable system.

As an example of single-source multiple ED architectures, $(PM_dM_eA)^N$ obtained from the renaming of (PM_dM_eA) complemented with a controller component is given below. $(PM_dM_eA)^N$ is a single-source multiple ED which is the composition of an interface, $IPMMA^N$, and N IMPs renamed as $(PM_dM_eA)_i$'s (Fig. 4.7). $IPMMA^N$ is itself the composition of a central controller (CC) and a status word consisting of N SW_i's which specify the status of N $(PM_dM_eA)_i$'s. Initially all the $(PM_dM_eA)_i$'s are idle. When all the $(PM_dM_eA)_i$'s are active, and there is an encoding or decoding request, CC stores this request but does not accept any more requests, until a $(PM_dM_eA)_i$ sends an encoding or decoding response. In this case, CC sends the stored request to this $(PM_dM_eA)_i$.

Multiple IMP Model for (PM_dM_eA)

$(PM_dM_eA)^N = IPMMA^N \| (\|_{i=1}^N (PM_dM_eA)_i)$

$IPMMA^N = CC \| (\|_{i=1}^N SW_{i,<i>})$

$CC = (ext?Drq \rightarrow sw.1!get \rightarrow ((\Box_{i=1}^N (sw.i?ready \rightarrow ext.i!DRq \rightarrow CC)) \Box (sw.N?noava \rightarrow CC_{<DRq>}))) \Box$

$(ext?ERq \rightarrow sw.1!get \rightarrow ((\Box_{i=1}^N (sw.i?ready \rightarrow ext.i!ERq \rightarrow CC)) \Box (sw.N?noava \rightarrow CC_{<ERq>}))) \Box$

$(\Box_{i=1}^N ((ext.i?DRe \rightarrow ext!DRe \rightarrow sw.i!idle \rightarrow CC) \Box (ext.i?ERe \rightarrow ext!ERe \rightarrow sw.i!idle \rightarrow CC)))$

$CC_{<DRq>} = (\Box_{i=1}^N (ext.i?DRe \rightarrow ext!DRe \rightarrow ext.i!DRq \rightarrow CC) \Box (ext.i?ERe \rightarrow ext!ERe \rightarrow ext.i!DRq \rightarrow CC))$

$CC_{<ERq>} = (\Box_{i=1}^N (ext.i?DRe \rightarrow ext!DRe \rightarrow ext.i!ERq \rightarrow CC) \Box (ext.i?ERe \rightarrow ext!ERe \rightarrow ext.i!ERq \rightarrow CC))$

$SW_{i,<i>} = (sw.i?get \rightarrow sw.i!ready \rightarrow SW_{i,<a>})$ $\quad 1 \le i \le N$

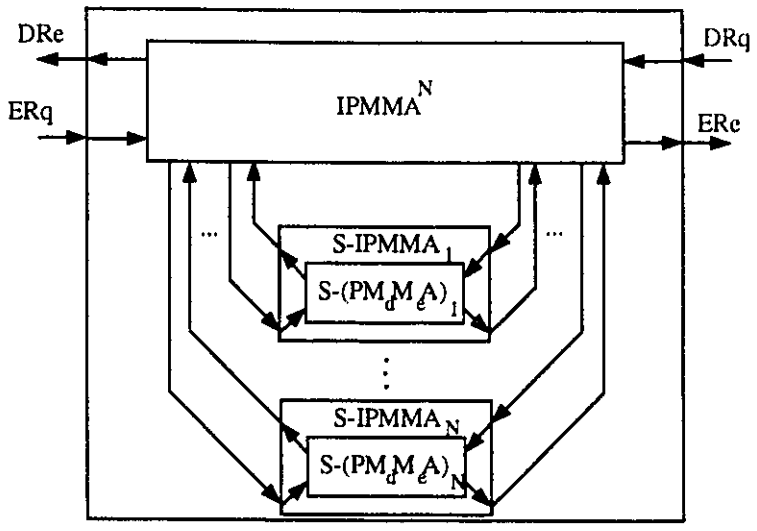
$SW_{i,<a>} = (sw.i?get \rightarrow sw.i+1!get \rightarrow SW_{i,<a>}) \Box (sw.i?idle \rightarrow SW_{i,<i>})$ $\quad 1 \le i < N$

$SW_{N,<a>} = (sw.N?get \rightarrow sw.N!noava \rightarrow SW_{N,<a>}) \Box (sw.N?idle \rightarrow SW_{N,<i>})$

$(PM_dM_eA)_i = f_i(PM_dM_eA)$

$\forall c.x \in \alpha(PM_dM_eA) \bullet f_i(c.x) = c.i.x$ $\quad 1 \le i \le N$

Figure 4.7 Single-Source $(PM_dM_eA)^N$ Model



Similar to previous multiple-layer and multiple-source models for (PM_dM_eA), $(PM_dM_eA)^N$ is not a hiding refinement for (PM_dM_eA). However, as in Lemma 4.1 and 4.2, we can show that $(PM_dM_eA)^N$ simulates (PM_dM_eA) when its environment simulates that of (PM_dM_eA).

Lemma 4.3 : $h^{-1}((PM_dM_eA)^N \| ENV)\backslash S_2 \sqsupseteq ((PM_dM_eA)\| ENV)$

where $S_2 = \alpha h^{-1}((PM_dM_eA)^N \| ENV) - \alpha((PM_dM_eA)\| ENV)$

$ENV = (ext!DRq \rightarrow ext?DRe \rightarrow ENV) \sqcap (ext!ERq \rightarrow ext?ERe \rightarrow ENV)$ and

$$h^{-1}(a) = \begin{cases} pmma.b & a = pmma.1.b \wedge (b = \mathcal{P}Rq \vee b = \mathcal{P}Re \vee b = \mathcal{M}_dRq \vee b = \mathcal{M}_dRe) \\ pmma.b & a = pmma.1.b \wedge (b = \mathcal{M}_eRq \vee b = \mathcal{M}_eRe \vee b = \mathcal{A}Rq \vee b = \mathcal{A}Re). \\ a & otherwise \end{cases}$$

Proof : It can be shown that

$h^{-1}(IPMMA^N \| ENV)\backslash S_2 = ENV$ and

$h^{-1}((\|_{i=1}^{N} f_i(PM_dM_eA)))\backslash S_2 = (PM_dM_eA)\|(\|_{i=2}^{N} f_i(PM_dM_eA)\backslash S_2)$. Therefore,

$h^{-1}((PM_dM_eA)^N \| ENV)\backslash S_2 = ((PM_dM_eA)\| ENV)$

since $\alpha ENV \cap \left(\bigcup_{i=2}^{N} \alpha(f_i(PM_dM_eA)\backslash S_2) \right) = \emptyset$. Since $P \sqsupseteq P$,

$h^{-1}((PM_dM_eA)^N \| ENV)\backslash S_2 \sqsupseteq ((PM_dM_eA)\| ENV)$. []

4.2.2 Multiple-Source Multiple EDs

Similar to single ED architectures, multiple EDs may be designed to serve multiple sources. Multiple ED architectures with multiple-sources are classified according to the encoding and decoding request sets of single EDs constituting the multiple ED architecture. Based on the assumption of homogeneoity, a multiple-source multiple ED is said to be dedicated if each ED_i is dedicated. If ED_i's of a multiple ED architecture are shared then the multiple ED architecture is also said to be shared.

4.2.2.1 Dedicated ED Model

As an example of multiple-source dedicated multiple ED architectures, MS_d-$(PM_dM_eA)^N$ is given below. Each $(PM_dM_eA)_i$ of MS_d-$(PM_dM_eA)^N$ serves a single encoding request and a decoding request source (Fig. 4.8).

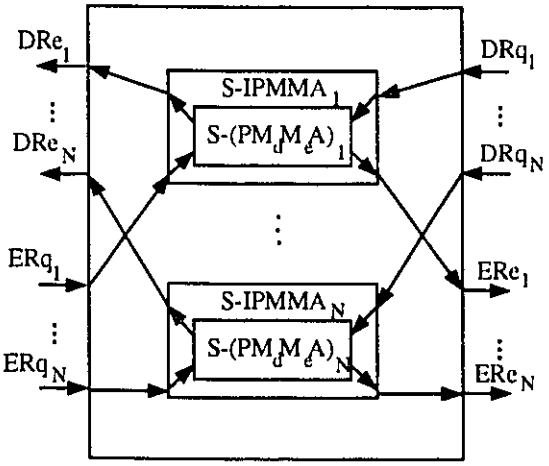
Multiple-Source (NxN) (Dedicated) $(PM_dM_eA)^N$ Description

$$MS_d\text{-}(PM_dM_eA)^N=(\|_{i=1}^{N}(PM_dM_eA)_i)$$

$$(PM_dM_eA)_i=f_i(PM_dM_eA)$$

$$\forall c.x\in\alpha(PM_dM_eA)\bullet f_i(c.x)=c.i.x \qquad 1\le i\le N$$

Figure 4.8 Multiple-Source (NxN) (Dedicated) $(PM_dM_eA)^N$ Model

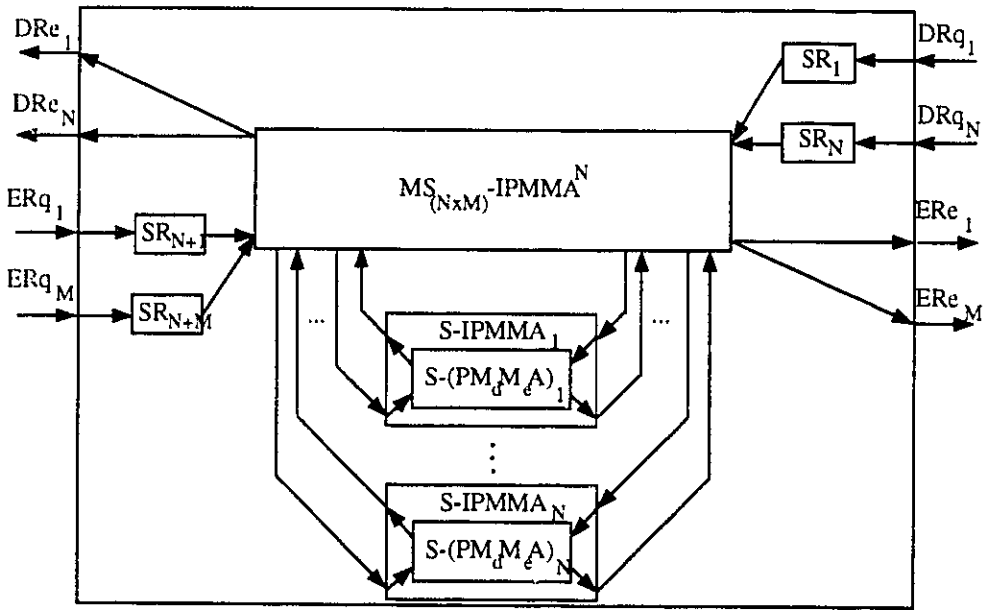


4.2.2.2 Shared ED Model

As an example of multiple-source shared multiple ED architectures, $MS_{(NxM)}$-$(PM_dM_eA)^N$ is given as a refinement of single-source $(PM_dM_eA)^N$. $MS_{(NxM)}$-$(PM_dM_eA)^N$ serves encoding requests coming from M sources and decoding requests coming from N sources in a cyclic fashion. Similar to multiple-source refinement for single ED given in Section 4.1.3, decoding requests are stored in registers $SR_1,...,SR_N$ and encoding requests

are stored in registers $SR_{N+1},...,SR_{N+M}$ (Fig. 4.9). In order to link the registers to the CC, a scheduler component, $MS_{(NxM)}$-SCH is added to the description of $(PM_dM_eA)^N$. The index of $MS_{(NxM)}$-SCH shows which source register is being checked. Initially, $MS_{(NxM)}$-SCH is ready to check SR_1 to see whether there is any decoding request from source 1.

Figure 4.9 Multiple-Source (NxM) (Shared) $(PM_dM_eA)^N$ Model



Multiple-Source (NxM) (Shared) $(PM_dM_eA)^N$ Description

$$MS_{(N\times M)}-(PM_dM_eA)^N=MS_{(N\times M)}-IPMMA^N\|(\|_{i=1}^{N}(MS_{(N\times M)}-PM_dM_eA)_i)\|(\|_{i=1}^{N+M}SR_{i,<>})$$

$$MS_{(N\times M)}-IPMMA^N=MS_{(N\times M)}-CC\|(\|_{i=1}^{N}SW_{i,<i>})\|MS_{(N\times M)}-SCH_1$$

$$MS_{(N\times M)}-CC=(\Box_{j=1}^{N}(ext?Drq_j\rightarrow sw.1!get\rightarrow((\Box_{i=1}^{N}(sw.i?ready\rightarrow ext.i!DRq_j\rightarrow MS_{(N\times M)}-CC))\Box$$
$$(sw.N?noava\rightarrow MS_{(N\times M)}-CC_{<DRq_j>}))))\Box$$
$$(\Box_{j=1}^{M}(ext?ERq_j\rightarrow sw.1!get\rightarrow((\Box_{i=1}^{N}(sw.i?ready\rightarrow ext.i!ERq_j\rightarrow MS_{(N\times M)}-CC))\Box$$
$$(sw.N?noava\rightarrow MS_{(N\times M)}-CC_{<ERq_j>}))))\Box$$
$$(\Box_{i=1}^{N}(\Box_{j=1}^{N}(ext.i?DRe_j\rightarrow ext!DRe_j\rightarrow sw.i!idle\rightarrow MS_{(N\times M)}-CC)))\Box$$
$$(\Box_{i=1}^{N}(\Box_{j=1}^{M}(ext.i?ERe_j\rightarrow ext!ERe_j\rightarrow sw.i!idle\rightarrow MS_{(N\times M)}-CC)))$$

$$MS_{(N\times M)}-CC_{<DRq_j>}=(\Box_{i=1}^{N}((\Box_{k=1}^{N}(ext.i?DRe_k\rightarrow ext!DRe_k\rightarrow ext.i!DRq_j\rightarrow MS_{(N\times M)}-CC))\Box$$
$$(\Box_{k=1}^{M}(ext.i?ERe_k\rightarrow ext!ERe_k\rightarrow ext.i!DRq_j\rightarrow MS_{(N\times M)}-CC))))$$

$$MS_{(N\times M)}-CC_{<ERq_j>}=(\Box_{i=1}^{N}((\Box_{k=1}^{N}(ext.i?DRe_k\rightarrow ext!DRe_k\rightarrow ext.i!ERq_j\rightarrow MS_{(N\times M)}-CC))\Box$$
$$(\Box_{k=1}^{M}(ext.i?ERe_k\rightarrow ext!ERe_k\rightarrow ext.i!ERq_j\rightarrow MS_{(N\times M)}-CC))))$$

$$MS_{(N\times M)}-SCH_i=(sr.i!check\rightarrow(sr.i?empty\rightarrow MS_{(N\times M)}\cdot\cdot SCH_{i+1})\Box \qquad 1\le i\le N$$
$$(sr.i?DRq\rightarrow ext!DRq_i\rightarrow MS_{(N\times M)}-SCH_{i+1}))$$

$$MS_{(N\times M)}-SCH_i=(sr.i!check\rightarrow(sr.i?empty\rightarrow MS_{(N\times M)}-SCH_{i+1})\Box \qquad N+1\le i<N+M$$
$$(sr.i?ERq\rightarrow ext!ERq_{i-N}\rightarrow MS_{(N\times M)}-SCH_{i+1}))$$

$$MS_{(N\times M)}-SCH_{N+M}=(sr.N+M!check\rightarrow(sr.N+M?empty\rightarrow MS_{(N\times M)}-SCH_1)\Box$$
$$(sr.N+M?ERq\rightarrow ext!ERq_M\rightarrow MS_{(N\times M)}-SCH_1))$$

$$(MS_{(N\times M)}-PM_dM_eA)_i=(\Box_{j=1}^{N+M}g_j(f_i(S-IPMMA)))\|f_i(S-(PM_dM_eA))$$

$$\forall c.x\in\alpha(PM_dM_eA)\bullet f_i(c.x)=c.i.x \qquad 1\le i\le N$$

$$\forall c.i.x\in\alpha f_i(PM_dM_eA)-\{ext.i.DRq,ext.i.ERq,ext.i.DRe,ext.i.ERe\}\bullet g_j(c.i.x)=c.i.x \qquad 1\le j\le N+M$$

$$g_j(ext.i.DRq)=ext.i.DRq_j \quad g_j(ext.i.DRe)=ext.i.DRe_j \qquad 1\le i\le N+M$$

$$g_j(ext.i.ERq)=ext.i.DRq_j \quad g_j(ext.i.ERe)=ext.i.ERe_j \qquad 1\le i\le N+M$$

Similar to the multiple IMP model for (PM_dM_eA), the multiple IMP model for $MS_{(NxM)}$-(PM_dM_eA, i.e. $MS_{(NxM)}$-$(PM_dM_eA)^N$ is not a hiding refinement for $MS_{(NxM)}$-(PM_dM_eA). However, as in Lemma 4.3, we can show that $MS_{(NxM)}$-$(PM_dM_eA)^N$ simulates $MS_{(NxM)}$-(PM_dM_eA) when its environment simulates that of $MS_{(NxM)}$-(PM_dM_eA).

Lemma 4.4 : $h^{-1}(MS_{(NxM)}-(PM_dM_eA)^N \| ENV_3)\backslash S_3 \sqsupseteq (MS_{(NxM)}-(PM_dM_eA)\| ENV_3)$

where

$S_3=\alpha h^{-1}(MS_{(NxM)}-(PM_dM_eA)^N \| ENV_3)-\alpha(MS_{(NxM)}-(PM_dM_eA)\| ENV_3)$,

$ENV_3=(\sqcap_{i=1}^{N}(ext.i!DRq \rightarrow ext?DRe_i \rightarrow ENV_3))\sqcap$

$(\sqcap_{i=N+1}^{N+M}(ext.i!ERq \rightarrow ext?ERe_{i-N} \rightarrow ENV_3))$ and

$$h^{-1}(a)=\begin{cases} pmma.b & a=pmma.1.b \wedge (b=\mathcal{P}Rq \vee b=\mathcal{P}Re \vee b=\mathcal{M}_dRq \vee b=\mathcal{M}_dRe) \\ pmma.b & a=pmma.1.b \wedge (b=\mathcal{M}_eRq \vee b=\mathcal{M}_eRe \vee b=\mathcal{A}Rq \vee b=\mathcal{A}Re). \\ a & otherwise \end{cases}$$

Proof : Similar to Lemma 4.3.

Now, having discussed various architectural models for PDU encoding and decoding, we can introduce the performance evaluation methodology for measuring the effects of such architectures on the end-to-end performance of protocol stacks.

Chapter 5 PERFORMANCE EVALUATION OF PDU ENCODERS / DECODERS

Since the basic motivation of developing a PDU encoding / decoding architecture based on a distributed implementation model is to achieve faster encoding and decoding, a mechanism is also needed to measure the effectiveness of the architecture in terms of speed-up against processing resources.

In the previous chapters, we have described a general framework for PDU encoding and decoding, a common distributed implementation model, and different architectural models to combine the phases of encoding and decoding. Each model employs different forms of parallelism, such as multiple execution units in modules, pipelined modules for encoding and decoding, and multiple modules for serving encoding / decoding requests of different sources from different layers. Therefore, the analysis of performances of these models requires different techniques if the *granularity level* of the analysis is chosen to be the components of an IMP. However, stand-alone performance figures for PDU EDs are not enough, since PDU EDs are integral parts of a possibly multi-layered communication architecture. Therefore, a performance analysis methodology should measure the effectiveness of PDU EDs in the framework of multi-layered communication architectures. The measure of effectiveness should be in terms of speed-up in the end-to-end communication of the multi-layered architecture against processing resources of the PDU ED.

In order to correctly analyze a communication architecture in terms of its performance, both the protocols employed and the physical resources on which they are implemented should be combined to model the architecture. In other words, a performance evaluation

methodology should capture the true picture of both the load, which is given as the formal specification of implemented protocols, and the implementation, which is a collection of resources of the communication architecture [Hec 91].

Usually, most analytical performance evaluation studies cover the analysis of specific mechanisms, or layers in isolation [Rei 82, Rei 86]. There are also models developed to analyze specific applications where a number of protocol layers – instead of a full stack – are taken into account [Fdi 90, Mit 86, Mur 88] as well as the methodologies to handle the full protocol stack where processing overheads due to each layer are considered [Con 88, Gih 86, Kri 86, Kue 86].

In this chapter, an overview of existing performance evaluation techniques for full protocol stacks is given. Two particular approaches [Con 88, Kri 86] are discussed in detail. A new methodology combining relative advantages of these approaches is introduced.

5.1 Overview of Full Protocol Stack Performance Models

Existing performance evaluation methodologies for full protocol stacks can be categorized into two main groups based on the concepts used to develop performance models. Performance models based on the queueing and traffic concepts are designed according to specific protocol functions and environments. On the other hand, performance models based on the formal specifications of protocols include different protocol functionalities. However, they have to be complemented with the information about the resources used to implement the protocols of the stack.

5.1.1 Performance Models based on Queueing and Traffic Concepts

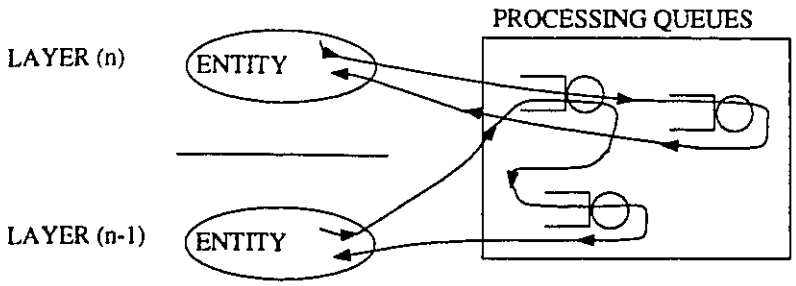
One of the early performance models for full protocol stacks is a successive hier-

archical decomposition and aggregation technique [Gih 86] which is proposed for the OSI model. This approach is based on constructing queueing submodels for each layer such that each submodel pictures the functions of the layer. The overall queueing model is solved by a decomposition and aggregation technique. In this model, connection establishment and release are assumed to have no effect on the end-to-end performance; hence all subnetworks are analyzed at the data transfer phase. The method requires high computational costs for large and complex communication architecture models.

A more recent approach for performance evaluation of full protocol stacks based on queueing and traffic concepts is the one proposed by Conway [Con 88]. The methodology includes developing a generic queueing network model based on the structure and specifications of the OSI model. In order to solve the resulting model, an iterative decomposition algorithm is used.

The generic model consists of a number of open systems with a number of layer entities which are interconnected through service access points (SAP's). The application layer entities are connected to finite closed sources. The SAP's correspond to flow control queues where functions such as segmenting / reassembling, concatenation / separation, blocking / deblocking and window flow control are performed. There is a set of processing queues which are shared by different entities and distributed over the open systems in the network. For each source, there is a closed routing chain which goes through a set of entities and processing queues. Overheads associated with the processing of a PDU by an entity are modeled by the visit of a set of processing queues by the PDU, as shown in Fig. 5.1. The set of processing queues may be shared among the entities of different layers.

Figure 5.1 Modeling of Processing Overheads [Con 88]

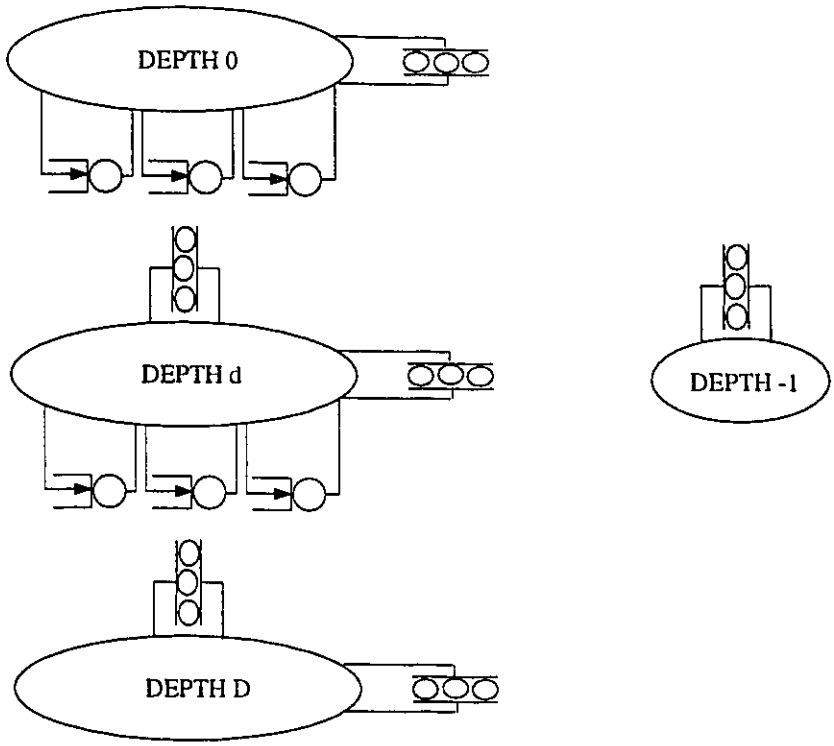


The overall network model is a multiple chain closed queueing network where a number of different scheduling policies for servers are available. The queueing model is not a product-form network, therefore an iterative decomposition algorithm is used to approximate the performance figures.

The iterative decomposition algorithm is based on the partitioning of a network into a set of subsystems. The subsystem of each particular queue is determined by the *depth* of a queue where the depth of a queue q_i is the maximum number of flow-control points that a chain has to pass through to reach queue q_i. The queues of the processing subsystem are assumed to be at depth –1. The number of depths in the model is assumed to be D.

A reduced network consisting of flow-equivalent queues for the part of the network which is complementary to the subsystem, is constructed for each subsystem as shown in Fig. 5.2. The flow-equivalent queues represent the delays of customers while traversing through complementary parts of the network.

Figure 5.2 Reduced Networks at Various Depths [Con 88]



Each reduced network is solved while the service time requirements at the flow-equivalent queues are updated to express the latest estimates of the delays in the complementary parts of the network. This procedure is repeated until convergence is obtained in the mean queue lengths of all the reduced networks or an iteration limit is exceeded.

The generic model is of a static nature, since the closed sources which feed the open connections are assumed to be always in the data transfer phase. Therefore, it cannot be used to model the dynamic nature of a system where connection establishment and release enclave the data transfer phase. In order to capture this dynamic nature, a "quasi-static approximation" is given in [Con 88] based on the assumption that the overheads involved in the process of connection establishment and release have a negligible effect on the conditional mean performance measures.

There are two shortcomings of this methodology. The first shortcoming is the use of quasi-static approximation to include the overheads associated with connection establishment and release; in particular, if the application layer protocol involves successive phases (regimes) to reach the data transfer phase, the basis of quasi-static assumption becomes rather weak. The other shortcoming is the difficulty of guaranteeing the accuracy and the convergence properties of the iterative algorithm; especially for large models it is quite cumbersome to validate the results with respect to simulation estimates.

There are two advantages of this methodology relative to other methodologies. The main advantage is the ability of analyzing complex protocol functionalities at different layers in a systematic way. The other advantage is the true reflection of coupling between layers both due to the interlayer traffic and the resource sharing.

5.1.2 Performance Models based on Formal Specifications

Another approach for performance evaluation of full protocol stacks is obtaining the performance model from the formal specifications of the protocols involved.

The approach of developing analytical performance models from the formal specifications of protocols originated in [Rud 83, Rud 84]. In this methodology, a Communicating Finite State Machine (CFSM) representation of a protocol is used to obtain the performance model. In the CFSM representation, the arcs of the *state-transition graph* are labelled with transition probabilities and delays. The resulting model is interpreted as a closed queueing network with a single customer. Although this methodology was originally proposed for single layers, a more recent approach [Kri 86] for performance evaluation of full stacks based on formal specifications is developed as an extension of this methodology and it is explained in detail below.

Another direction in the derivation of performance models based on formal specifi-

cations is timed Petri net models. In [Mol 82], Stochastic Petri Nets (SPN) are shown to be a useful tool for performance evaluation. In a SPN, mean firing rates are assigned to the transitions of the Petri net model obtained for the protocol being evaluated. The marking of the Petri net model is defined as the ordered n-tuple of the number of tokens in the n places of the Petri net. When the firing rates have exponential distributions, the change of the SPN from one marking to the next becomes isomorphic to the behaviour of a continuous-time homogeneous Markov chain [Mol 82]. The SPN approach is shown to be equivalent to the performance evaluation models based on CFSMs [Kri 86].

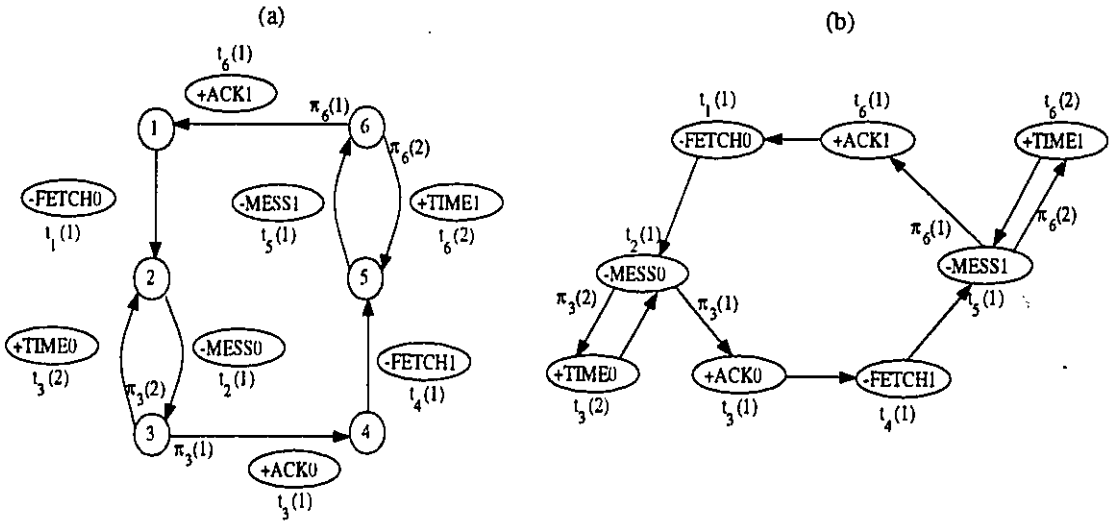
Another timed formal specification model is a timed Extended FSM (EFSM) model [Sha84] which is proposed for the verification of protocols. The EFSM model allows multiple state variables of different types such that the state becomes a vector of these variables and state transition becomes rather complex functions of these variables. In the timed EFSM model, the discrete-valued timer variables are used to measure the elapse of time and time events. The time variables and time events are local to each protocol entity in the EFSM model. Time intervals are associated with events which occur in the model. Although it is possible to use the timed EFSM models for performance evaluation, due to the use of time intervals, they are better suited for verification purposes.

In [Kri 86], the network model consists of a single open system. In each layer, there are a number of open connections for which only the local entity is modeled explicitly. Each entity is associated with a state-transition graph for the FSM which specifies the behaviour of the entity according to the protocol.

The state-transition graph for an entity is a labeled directed graph whose vertices represent the states of the FSM and the labels of edges describe the expected duration and probability of transitions. Fig. 5.3.a shows the state-transition graph for an alternating bit protocol entity. The state-transition graph is converted into an equivalent representation

called the *transition-relation graph* in which the vertices correspond to edges in the state-transition graph. In the transition-relation graph, delays are associated with states, and state transition is assumed to be instantaneous. Fig. 5.3.b shows the transition-relation graph corresponding to Fig. 5.3.a.

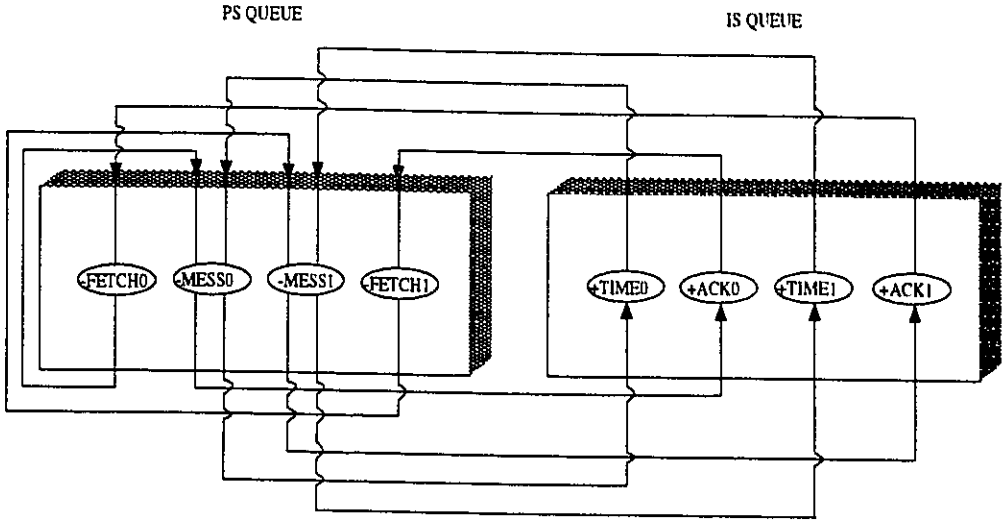
Figure 5.3 Alternating Bit Protocol : (a) State-transition Graph, (b) Transition-relation Graph [Kri 86]



In this methodology, the states of the transition-relation graph are divided as active states which are labeled (-) and passive states which are labeled (+). It is assumed that active states require execution at a local processing unit, whereas passive states are associated with delays of waiting for a response from the remote entity. Based on this concept, the transition-relation graph is rearranged such that all active states are assigned to a processor sharing (PS) queue and all passive states are assigned to an infinite server (IS) queue. Fig. 5.4 shows the queueing network model obtained from the transition-relation graph of Fig. 5.3.b.

The resulting model becomes a closed two queue single chain queueing network with class switching which has a single customer. It is shown that such a queueing network can be solved by using computational algorithms for product-form queueing networks [Lav 83].

Figure 5.4 Queuing Network Model for Fig. 5.3.b [Kri 86]



In order to obtain the model for multiple entities at multiple layers, a multiple chain closed queueing network is constructed. In this model, there is a closed chain with a single customer for each entity at every layer of the open system. The resulting model is also a closed product-form queueing network.

There are certain shortcomings of this methodology. The first shortcoming is the modeling of only the local system and the assumption of the remote system as a delay center. Modeling of the remote system as a delay center excludes the modeling of resource sharing among layers at the remote system. Another shortcoming is the exclusion of the interdependency between adjacent layers in terms of the traffic flow. The model permits only a single PS type resource; therefore it cannot be used to model the effects of a number of resources for different protocol functionalities.

The main advantage of this methodology is its ability of reflecting a more detailed picture of the processing requirements of different protocol functionalities than methodologies based on the queueing and traffic concepts. This feature is especially important to model the behaviour of systems during the transient phases of communication, namely connection establishment and release.

5.1.3 Proposed Performance Evaluation Methodology

The encoding / decoding architectures discussed in chapter 4 are special purpose components used to serve one or a number of layers in a protocol stack. Since our goal is to analyze the effects of such components on the end-to-end performance of the whole protocol stack, it is necessary to use a performance evaluation methodology involving the entire stack.

The effects of the inclusion of such special purpose components are more pronounced on the layers they serve since they cause the redistribution of processing overheads of entities of these layers. However, they also have relatively smaller effect on the other layers of the protocol stack due to the interlayer traffic flow.

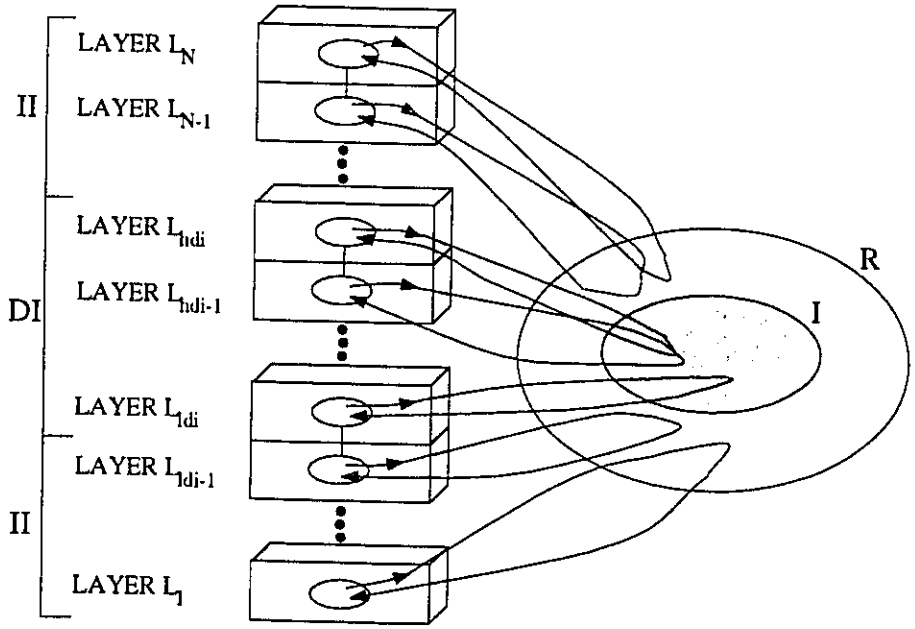
As discussed in previous subsections, performance models based on formal specifications are more powerful in expressing the details of the processing overheads. On the other hand, performance models based on queueing and traffic concepts have the advantage of analyzing protocol stacks in a more systematic way. In order to combine these relative advantages while measuring the effects of special purpose components on the end-to-end performance, layers of the protocol stack may be divided into two categories : *direct interest* (DI) layers and *indirect interest* (II) layers.

The DI layers are the layers whose entities use the special purpose component, in our case the PDU ED, whose effects on the end-to-end performance are investigated. All the other layers of the protocol stack constitute the II layers. In order to obtain a detailed analysis of the processing overhead incurred at the DI layers, a performance evaluation methodology based on formal specifications is used. On the other hand, the queueing model for the II layers are constructed using a methodology based on the queueing and traffic concepts. These two models are solved together iteratively to obtain the end-to-end

performance of the entire protocol stack. The methodology is applicable for measuring the effects of any special-purpose component on the end-to-end performance of the layered communication architecture.

In Fig. 5.5, the partition of layers of a layered communication system based on the use of resources is shown. According to the figure, only the DI layers use the resources in the set I which is a subset of the set of resources R for the communication system. Therefore, the partition of the layers is based on the resources in the set I. The *highest direct interest* (hdi) layer is L_{hdi}, whereas the *lowest direst interest* (ldi) layer is L_{ldi} according to the figure.

Figure 5.5 Partition of Layers of a Layered Communication System



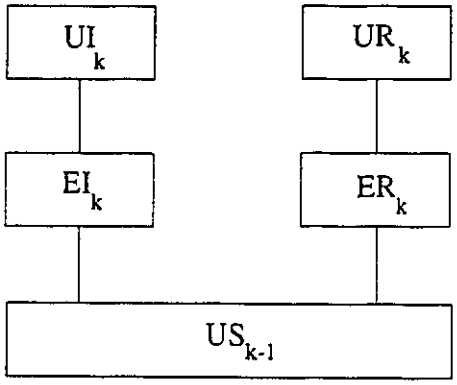
The proposed performance evaluation methodology consists of successive steps. The first step is the partition of layers according to the use of a set of specific resources whose effects on the end-to-end performance are investigated.

The second step involves the generation of the queueing network model for the DI layers based on the extension of the performance evaluation methodology of [Kri 86].

The extensions provide the inclusion of multiple queues at both local and remote systems and the inclusion of the dependencies between adjacent layers due to the traffic flow.

Each DI layer is modeled by two entities, an *initiator* and a *responder*; two users, an *initiator user* and a *responder user*; and an *underlying service*, as shown in Fig. 5.6. Entities, users, and underlying service are modeled by Communicating Finite State Machines (CFSMs).

Figure 5.6 DI Layer FSMs



In order to obtain the queueing network model for the DI layers, first a transition-relation graph which is called the *abstract transition-relation graph* (ATRG), that represents the behaviour of all DI layers, is generated. In the ATRG, the vertices represent the send and receive transitions of CFSMs of the DI layers and the labels of edges describe the probability of transitions. The generation of the ATRG starts at layer L_{hdi}. A transition-relation graph is generated using a modified version of the perturbation algorithm for protocol validation [Wes 78]. Next, another transition-relation graph is generated for the layer below and the resulting graphs are linked together to obtain a single graph. Linking of the graphs for two adjacent layers L_k and L_{k-1} eliminates all vertices associated with US_{k-1}, UI_{k-1} and UR_{k-1}. This process is repeated until the ATRG is linked with the transition-relation graph of layer L_{ldi}. The resulting graph $ATRG_{hdi,ldi}$ contains vertices

for all send / receive transitions of entities of the DI layers, users of layer L_{hdi}, and the underlying service for layer L_{ldi}.

In order to model the processing overhead associated with send / receive transitions, vertices of the $ATRG_{hdi,ldi}$ are replaced with subgraphs. Therefore another transition-relation graph which is called the *detailed transition-relation graph* ($DTRG_{hdi,ldi}$) is obtained. In the $DTRG_{hdi,ldi}$, each send and receive transition is modeled as a tour of the PDU through the set of resources. These tours represent the processing overheads involved in detail.

The $DTRG_{hdi,ldi}$ is converted into a single-chain closed queueing network consisting of queues corresponding to resources used by the DI layers.

Scheduling policies of resources are assigned according to resource characteristics. The granularity level in the performance model, i.e. what constitutes a resource, is based on the minimization of the complexity of the resulting queueing network while maximizing the amount of detail in the processing overhead modeling. In the modeling of PDU EDs, each IMP is chosen as a resource. The job characteristics of the resources are obtained through techniques such as *benchmarking*, *simulation*, or using estimations.

In the third step of the methodology, the II layers are modeled by using the methodology based on queueing and traffic concepts [Con 88]. These two models are solved together iteratively.

The effectiveness of the PDU EDs is measured in terms of the increase in the amount of data transferred per unit end-to-end connection time against the total number of execution units in the PDU ED.

5.2 Partition of Layers

The first step of the performance evaluation methodology is the partition of layers according to the use of a resource subset. In order to explain this partition, let us formalize the communication systems in terms of layers and resources.

An *open system* O is a three-tuple (S,R,P) where *protocol stack* S=(L_1,...,L_N) is an N-tuple of layers, R={r_1,...,r_m} is a set of *resources* and P is the *load function* of S on R, such that if $\rho_{i,j}$=1, then r_j is a resource for L_i. If L_i does not use the resource r_j, $\rho_{i,j}$ is 0.

Definition 5.2.1: Given an open system $O = (S, R, \mathrm{P})$, the set of resources for layer L_i is $R_{L_i} = \{r_j | r_j \in R \bigwedge \rho_{i,j} = 1\}$ where the use of resources is continuous such that if $r_j \in R_{L_{i-1}} \bigwedge r_j \in R_{L_{i+1}}$, then $r_j \in R_{L_i}$.

The layers of S can be divided into two parts according to their use of a given resource subset I in R. This division is needed to decide which performance evaluation methodology should be used to generate the queueing model for a certain layer.

Definition 5.2.2: Given an open system $O = (S, R, \mathrm{P})$ and a resource subset I where $I \subseteq R$, layer L_{ldi} is the *lowest layer of direct interest* for I, iff $R_{L_{ldi}} \cap I \neq \emptyset \bigwedge \forall j < ldi \bullet R_{L_j} \cap I = \emptyset$. Similarly, layer L_{hdi} is the *highest layer of direct-interest* for I, iff $R_{L_{hdi}} \cap I \neq \emptyset \bigwedge \forall j > hdi \bullet R_{L_j} \cap I = \emptyset$.

Based on the definition of L_{ldi}, and L_{hdi}, we can define the partition of layers.

Definition 5.2.3: Given an open system $O = (S, R, \mathrm{P})$ and a resource subset I where $I \subseteq R$, a layer $\mathrm{L_i}$ is a *layer of direct-interest* (DI) for I iff $ldi \leq i \leq hdi$. A layer $\mathrm{L_j}$ is a *layer of indirect-interest* (II) for I iff $j < ldi \vee j > hdi$.

5.3 Direct Interest Layer Modeling

The second step of the performance evaluation methodology is the modeling of the DI layers. In order to obtain the queueing model for the DI layers, we first have to introduce the CFSM model for the DI layers.

5.3.1 CFSM Model for the DI Layers

Previous performance evaluation methodologies based on formal specifications do not include the underlying service between the sender and the receiver entities while modeling layers. Therefore, according to these methodologies it is not possible to construct a transition-relation graph for multilayered and multi-entity systems where interlayer traffic is taken into account. In order to obtain the performance evaluation model for two FSM's which communicate through other FSM's modeling the channel (service) between them, a more general approach is required. Such an approach includes the stepwise construction of the global transition-relation graph for multilayered systems.

Before describing the method of transition-relation graph derivation, let us give some definitions for CFSMs as well as some assumptions about the CFSM network to be used to model a DI layer.

Definition 5.3.1: A CFSM C_i is a four-tuple (σ_i,M_i,δ_i,q_i) of the following components :

1. A non-empty finite state set σ_i,
2. A finite message set M_i,
3. A partial transition function $\delta_i : \sigma_i \text{ X } M_i \rightarrow \sigma_i$,
4. A designated element q_i in σ_i as the initial state.

The message set of C_i is the union of the input message set $+M_i$ and the output message set $-M_i$, i.e. $M_i = +M_i \bigcup -M_i$. Based on this partition, we can classify the states of a CFSM according to the characteristics of the transitions originating from these states.

Definition 5.3.2: Let $C_i = (\sigma_i, M_i, \delta_i, q_i)$ be a CFSM and a given state $s \in \sigma_i$.

1. If $\forall m \in M_i \bullet \delta_i(s,m) = \emptyset$, then s is a *final state* of C_i,
2. If $(\forall m \in +M_i \bullet \delta_i(s,m) = \emptyset) \bigwedge (\exists m \in -M_i \bullet \delta_i(s,m) \neq \emptyset)$, then s is a *sending state* of C_i,
3. If $(\forall m \in -M_i \bullet \delta_i(s,m) = \emptyset) \bigwedge (\exists m \in +M_i \bullet \delta_i(s,m) \neq \emptyset)$, then s is a *receiving state* of C_i,
4. If $(\exists m \in -M_i \bullet \delta_i(s,m) \neq \emptyset) \bigwedge (\exists m \in +M_i \bullet \delta_i(s,m) \neq \emptyset)$, then s is a *mixed state* of C_i.
5. If s is a sending state and $\forall m \in M_s \bullet \delta_i(s,m)$ where $M_s \subseteq M_i$, then $\sum_{m \in M_s} prob(\delta_i(s,m)) = 1$.

Now based on definition 5.3.1, we can define a DI layer L_i as a collection of five CFSMs, namely user-initiator (UI_i), entity-initiator (EI_i), underlying-service (US_{i-1}), entity-responder (ER_i), and user-responder (UR_i). In the following definition, for given two CFSMs C_i and C_j, $M_{ij} = \{m | m \in -M_i \wedge m \in +M_j\}$ is the set of messages C_i sends to C_j; whereas $M_{ji} = \{m | m \in +M_i \wedge m \in -M_j\}$ is the set of messages C_i receives from C_j.

Definition 5.3.3: A DI layer $L_i =< UI_i, EI_i, US_{i-1}, ER_i, UR_i >$ is a four-tuple $(\Sigma_i, \mathrm{M}_i, \Delta_i, Q_i)$ of the following components :

1. A non-empty finite state set $\Sigma_i \subseteq (\sigma_{UI_i} \mathrm{x} \sigma_{EI_i} \mathrm{x} \sigma_{US_{i-1}} \mathrm{x} \sigma_{ER_i} \mathrm{x} \sigma_{UR_i})$,
2. A finite message set M_i where $\mathrm{M}_i = \bigcup_{j,k=UI_i,EI_i,US_{i-1},ER_i,UR_i} M_{jk}$ such that
   - $M_{jk} \bigcap M_{lm} = \emptyset$ if (jk) $\neq$ (lm),
   - $M_{jj} = \emptyset$,
   - $M_{UI_i US_{i-1}} = M_{US_{i-1} UI_i} = M_{UR_i US_{i-1}} = M_{US_{i-1} UR_i} = \emptyset$,
   - $M_{EI_i UR_i} = M_{UR_i EI_i} = M_{ER_i UI_i} = M_{UI_i ER_i} = \emptyset$,
   - $M_{EI_i ER_i} = M_{ER_i EI_i} = \emptyset$,
3. A binary relation Δ_i between states,
4. An initial state $Q_i = (q_{UI_i}, q_{EI_i}, q_{US_{i-1}}, q_{ER_i}, q_{UR_i})$.

According to definition 5.3.3, only machines in pairs <UI_i,EI_i>, <EI_i,US_{i-1}>, <US_{i-1},ER_i>, <ER_i,UR_i>, and <UI_i,UR_i> communicate with each other. For each send transition of an FSM, there is one and only one corresponding receive transition in a connected FSM.

Note that according to definition 5.3.3 UI_i and UR_i may directly communicate. This communication must be used for modeling purposes only. The idea is to provide "virtual acknowledgements" to data transfer phase PDUs without introducing load into the system. The same technique is used in performance modeling where flow control schemes such as selective-repeat are considered [Con 88, Lam 76].

In definition 5.3.3, the states of layer L_i are defined as five-tuples consisting of states of individual CFSMs. There are no messages in transit, i.e. queued messages in the definition of states, because it is assumed that when a CFSM C_i sends a message m to another CFSM C_j, C_j is always at a state ready to receive m. Based on this assumption,

global state transitions always correspond to state changes only in two CFSMs. Other assumptions about the states of layers are given below. In the following assumptions, k is the index of any CFSM of L_i whereas s and $s^{'}$ are states of CFSMs.

i) $\forall s \in \sigma_k \bullet \neg (s \; is \; a \; mixed \; state.)$

ii) $\forall s \in \sigma_k, m \in M_k \bullet \left(\delta_k(s, +m) = s^{'} \Rightarrow \neg \left(s^{'} \; is \; a \; receiving \; state.\right)\right)$

iii) $\forall s \in \sigma_k, m \in M_k \bullet \left(\delta_k(s, -m) = s^{'} \Rightarrow \neg \left(s^{'} \; is \; a \; sending \; state.\right)\right)$

iv) q_{UI_i} is a sending state, whereas $q_{EI_i}, q_{US_{i-1}}, q_{ER_i}$ and q_{UR_i} are receiving states.

Now, based on these assumptions about the states of CFSMs of layer L_i, we can define the binary relation Δ_i.

Definition 5.3.4: The global state $S^{'}$ of layer $L_i$is a next state of S, i.e. $S^{'} = \Delta_i(S)$, iff there exist i, j such that $_{ij=UI_iEI_i \vee EI_iUS_{i-1} \vee US_{i-1}ER_i \vee ER_iUR_i \vee UI_iUR_i}$ satisfying one of the following two conditions:

1. All the elements of $S^{'}$ and S are equal except
$\delta_i(s_i, -m) = s_i^{'} \bigwedge \delta_j(s_j, +m) = s_j^{'}$.
2. All the elements of $S^{'}$ and S are equal except
$\delta_i(s_i, +m) = s_i^{'} \bigwedge \delta_j(s_j, -m) = s_j^{'}$.

The reflexive and transitive closure of Δ_i is Δ_i^* which is used to define the reachable states from the initial state Q_i.

The definition of Δ_i does not restrict it to events happening one at a time instead of in parallel. If two pairs of CFSMs engage in communication simultaneously, then they can be represented as happening one after the other in any order.

Now based on the definition of the DI layers, we can introduce the algorithms for ATRG generation.

5.3.2 Abstract Transition-Relation Graph Generation

The abstract transition-relation graph $ATRG_i$ for layer L_i has the vertices corresponding to send or receive transitions of CFSMs of L_i, and the edges are labeled with the probabilities of the next transitions. According to definition 5.3.3 and further assumptions about layer L_i, each vertex corresponding to a send transition is connected to another vertex corresponding to a receive transition. The edge connecting such two vertices is labeled with probability 1, since there is no other possible transition for a send transition. On the other hand, a vertex corresponding to a receive transition may be connected to many vertices corresponding to send transitions. $ATRG_i$ for layer L_i can be generated by a modified version of the perturbation algorithm for protocol validation [Wes 78].

In algorithm 5.1, the set D is used to store the L_i's states whose perturbations are done, and the set N is used to store the states to be added to the state set of L_i. Initially, the $ATRG_i$ for layer L_i does not have any vertices, and the only state in Σ_i is the initial state Q_i. In the second step for each global state S_j, whose perturbation is not done, the set of next global states N_{S_j} is computed. If S_j is not a final state, i.e. $N_{S_j} \neq \emptyset$, for each global state S_k in N_{S_j}, two new vertices : $s_{k,s}$ for the send transition and $s_{k,r}$ for the receive transition are added to the $ATRG_i$. Algorithm 5.1 stops when there is no element in N whose perturbation is not done. In algorithm 5.1, the notation $pro\left(\delta\left(s_{j,r}\right) = s_{k,s}\right) = p_{j,k}$ is used to denote the edge between the vertices $s_{j,r}$ and $s_{k,s}$ and is labeled by the probability $p_{j,k}$.

The vertices of the abstract transition-relation graph $ATRG_i$ are partitioned into sets of vertices for each CFSM of L_i. In other words,
$ver\left(ATRG_i\right) = S_{UI_i} \cup S_{EI_i} \cup S_{US_{i-1}} \cup S_{ER_i} \cup S_{UR_i}$.

Algorithm 5.1 :

Step1. $\Sigma_i = \{Q_i\}, D = \{\}, N = \{\}, ver(ATRG_i) = \{\}$.

Step2. $\forall S_j \in \Sigma_i$ **do**

{**If** $S_j \notin D$ **then**

{$D = D \cup \{S_j\}$.

Compute $N_{S_j} = \{S_k | \Delta_i(S_j) = S_k\}$.

If $N_{S_j} \neq \emptyset$ **then**

{$\forall S_k \in N_{S_j}$ **do**

{$ver(ATRG_i) = ver(ATRG_i) \cup \{s_{k,s}, s_{k,r}\}$.

$pro(\delta(s_{k,s}) = s_{k,r}) = 1$.

$pro(\delta(s_{j,r}) = s_{k,s}) = p_{j,k}$.}

$N = N \cup N_{S_j}$.}}}

Step3. **If** $N \subseteq D$ **then** {Stop.}

else {$\Sigma_i = \Sigma_i \cup N, N = \{\}$. Goto Step 2.}

In order to obtain the $ATRG_{hdi,ldi}$ for all DI layers from L_{hdi} to L_{ldi}, algorithm 5.1 should be used in an incremental way from top to bottom where, at each step, the transition-relation graph $ATRG_i$ for layer L_i is generated and then linked to the transition-relation graph $ATRG_{hdi,i+1}$ generated for the above layers by replacing the vertices in the set S_{US_i} with the vertices in the sets $S_{EI_i}, S_{US_{i-1}}, S_{ER_i}$.

Algorithm 5.2 :

Step1. Use algorithm 5.1 to generate $ATRG_{hdi}$. $ATRG_{hdi,hdi} \leftarrow ATRG_{hdi}, i = hdi$.

Step2. $i = i - 1$.

If $i < ldi$ **then** {Stop.}

Step3. Use algorithm 5.1 to generate $ATRG_i$.

Step4. $ver(ATRG_{hdi,i}) \leftarrow (ver(ATRG_{hdi,i+1}) - S_{US_i}) \bigcup (ver(ATRG_i) - S_{UI_i} - S_{UR_i})$.

Step5. $\forall s_{j,s} \in (ver(ATRG_{hdi,i+1}) - S_{US_i})$ **do**

{Find the vertex $s_{k,r} \in (ver(ATRG_i) - S_{UI_i} - S_{UR_i}) \bullet s_{j,s}$ and $s_{k,r}$ are associated with the transmission of the same message.

$pro(\delta(s_{j,s}) = s_{k,r}) = 1$.}

Step6. $\forall s_{j,r} \in (ver(ATRG_{hdi,i+1}) - S_{US_i})$ **do**

{Find the vertex $s_{k,s} \in (ver(ATRG_i) - S_{UI_i} - S_{UR_i}) \bullet s_{j,s}$ and $s_{k,r}$ are associated with the transmission of the same message.

$pro(\delta(s_{k,s}) = s_{j,r}) = 1$.}

Step7. Go to step 2.

Algorithm 5.2 results in the $ATRG_{hdi,ldi}$ which shows the state of a single connection between the initiator and the responder users. In the $ATRG_{hdi,ldi}$, each vertex is associated with the sending or receiving of a message, i.e. a PDU. In order to obtain the $DTRG_{hdi,ldi}$, the vertices of the $ATRG_{hdi,ldi}$ should be replaced by the details of the processing overhead for sending and receiving of PDUs.

5.3.3 Detailed Transition-Relation Graph Generation

The vertices of the transition-relation graph $ATRG_{hdi,ldi}$, obtained for the DI layers, are associated with send or receive transitions at different layers. This model is directly related to the logical behavior of included layers, however the real performance model can

only be obtained when the processing overhead associated with each vertex is distributed on the resources on which the protocol layers are implemented.

The set $\mathcal{D}$ includes the vertices for which processing overhead is analyzed in detail. When there is an II layer above layer L_{hdi} in the protocol stack, the processing overhead for the vertices associated with entities of the DI layers are detailed in the $DTRG_{hdi,ldi}$, i.e. $\mathcal{D}=ver\left(ATRG_{hdi,ldi}\right) - S_{US_{ldi-1}} - S_{UI_{hdi}} - S_{UR_{hdi}}$. When there is no II layer above layer L_{hdi} in the protocol stack, the processing overhead for the vertices associated with entities of the DI layers as well users of layer L_{hdi} are detailed in the $DTRG_{hdi,ldi}$, i.e. $\mathcal{D}=ver\left(ATRG_{hdi,ldi}\right) - S_{US_{ldi-1}}$.

Processing overhead (PO) associated with each transition $s \in \mathcal{D}$ can be modeled as a collection of tasks. A task is a set of operations which are performed on some input data and is atomic. A major aspect in modeling the PO is the construction of precedence relationships among the tasks of the PO, i.e. the partial order of task execution. In order to model the tasks, we assume that there are three types of precedence relationships:

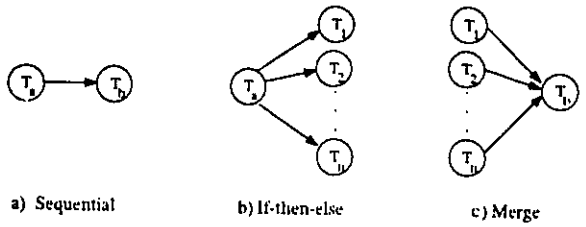
a) *Sequential relationship :* S : $T_a \rightarrow T_b$ specifies that for any PO to include these two tasks, task T_a must be completed before task T_b may begin. Once T_a is completed, task T_b is executed.

b) *If-then-else relationship :* IF : $T_a \rightarrow \boldsymbol{T_I}$ specifies that for any PO to include task T_a and the tasks in the set $\boldsymbol{T_I}$, task T_a must be completed before any task in $\boldsymbol{T_I}$ may begin. Once T_a is completed, one and only one task in $\boldsymbol{T_I}$ is selected to be executed according to some selection procedure.

c) *Merge relationship :* M : $\boldsymbol{T_I} \rightarrow T_b$ specifies that for any PO to include the tasks in the set $\boldsymbol{T_I}$ and task T_b, one task in $\boldsymbol{T_I}$ must be completed before task T_b may begin. For any such PO, only one task in $\boldsymbol{T_I}$ is executed. Once this execution is completed, task T_b is executed.

A graphical notation to represent different types of precedence relationships is given in Fig. 5.7. In this notation, nodes represent tasks, and the directed arcs among nodes represent the order of execution of the tasks. A graph representing the structure of a PO is called the *structure graph* of PO.

Figure 5.7 Models of Precedence Relationships between Tasks



All POs are assumed to have acyclic structure graphs such that there exists a non-empty set of final tasks T_F which do not have any succeeding tasks and an imaginary initial task T_0 with no processing requirements. There exists a path in the structure graph from T_0 to each T_f in T_F.

Consider a PO associated with a DI layer L_i consisting of a collection of n tasks $\{T_j \mid j=1,..,n\}$. For each task T_j, there is one and only one resource $r_k \in R_{L_i}$. A resource r_k may be used to implement a number of tasks. Once a resource has begun execution of a task, it completes this instance of task execution without any further communication with other resources. It is assumed that the selection of the next task in an IF relationship is determined through a fixed distribution and is independent of previous selections. The load of a task T_j on a resource r_k is defined by the mean service-time of the task on the resource, $1/\mu_{j,k}$.

Based on the properties of the structure graphs, the $ATRG_{hdi,ldi}$ can be converted into another transition-relation graph $DTRG_{hdi,ldi}$ whose vertices are labeled with the associated resource and the characteristics of the processing time of the task on the

resource. The conversion is based on the replacement of vertices of the $ATRG_{hdi,ldi}$ by their task graphs.

5.3.4 Queueing Network for the DI Layers

The vertices of the detailed transition-relation graph $DTRG_{hdi,ldi}$ are isomorphic to *classes* of *jobs* at the processing queues representing the resources.

The scheduling policies for processing queues include First-Come First-Served (FCFS), Last-Come First-Served Preemptive Resume (LCFSPR), Processor Sharing (PS), all with a single server or Infinite Servers (IS). FCFS queues are either of BCMP type, where BCMP [Bas 75] denotes a queue of the quasi-reversible type, or non-BCMP type. If an FCFS queue is of the BCMP type, then the service requirements of all class of customers visiting the queue must be equal and exponentially distributed.

According to the methodology of [Con 88], a closed queuing network model is used for the II layers. The closed queueing network model is chosen to circumvent the difficulty of asserting the stableness of queueing networks with open sources. In order to facilitate the consistency between the models for the DI and II layers, the closed queueing network model is also used for the DI layers.

The characteristics of different classes of jobs, and processing queues are needed for each resource. For PDU EDs, an IMP is considered as a resource, and the service time characteristics for encoding and decoding different PDUs involved in the communication are obtained through either measurement techniques such as benchmarking and simulation, or by using estimates.

5.4 Indirect Interest Layer Modeling

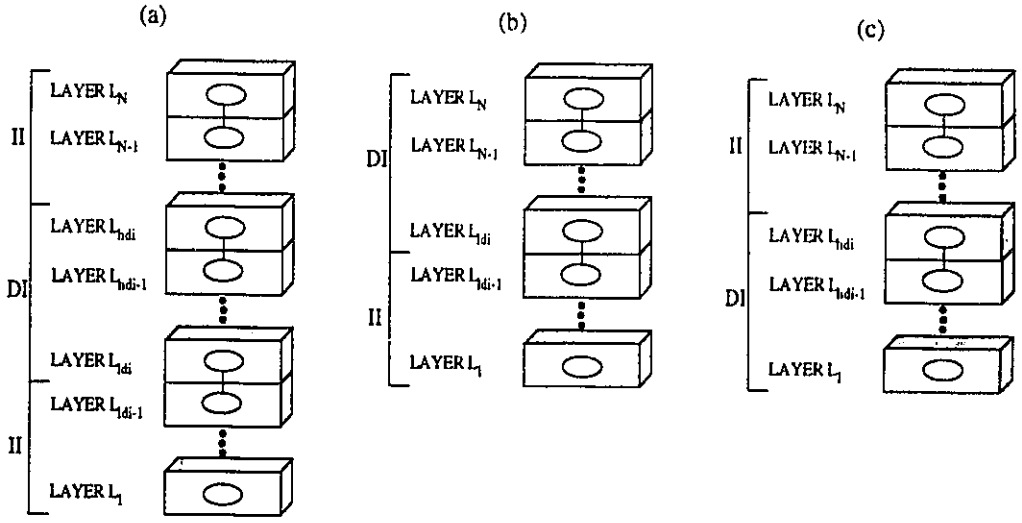
According to the partition of layers based on the use of a set of resources, there may be II layers above layer L_{hdi}, and below layer L_{ldi}. A degenerate case of partition occurs when all the layers use the resources in the resource set I. In this case, the highest and the lowest direct interest layers become layer L_N and layer L_1, respectively. However, when the resource set is used by a number of layers instead of the entire stack, there must be some II layers. There are three cases of partition when there are II layers :

a) Layers L_N to L_{hdi+1} are upper II layers, layers L_{hdi} to L_{ldi} are the DI layers, layers L_{ldi-1} to L_1 are lower II layers; when $N > hdi$ and $ldi > 1$ (Fig. 5.8.(a)).

b) Layers L_N to L_{ldi} are the DI layers, layers L_{ldi-1} to L_1 are lower II layers; when $N = hdi$ and $ldi > 1$ (Fig. 5.8.(b)).

c) Layers L_N to L_{hdi+1} are upper II layers, layers L_{hdi} to L_1 are the DI layers; when $N > hdi$ and $ldi = 1$ (Fig. 5.8.(c)).

Figure 5.8 Cases of Layer Partition

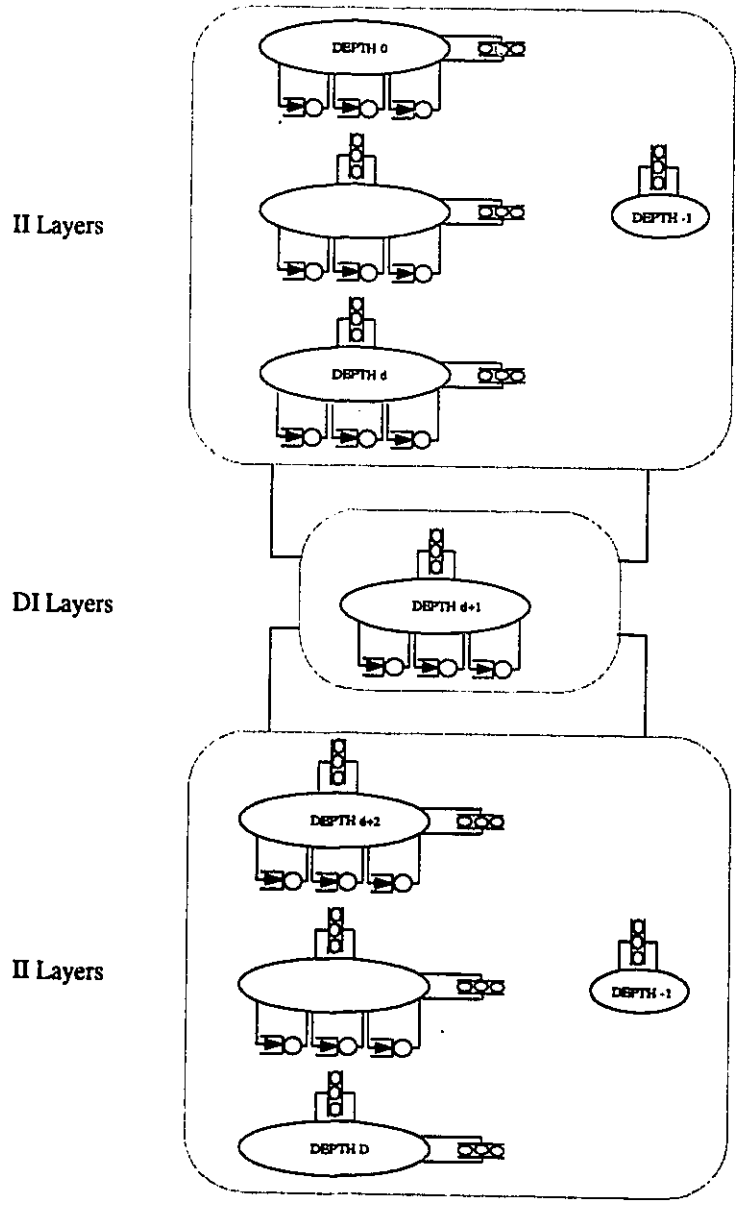


Both the upper and the lower II layers are modeled according to the performance evaluation methodology discussed in subsection 5.1.1. The upper and the lower II layers

are partitioned separately to obtain the reduced networks. The reduced networks are linked to the queueing network obtained for the DI layers.

Based on the mode of the partition of layers, a global queueing network model, as the collection of the queueing models of the DI and II layers, is obtained. Fig. 5.9 shows the global queueing network when there are both the upper and the lower II layers.

Figure 5.9 Global Queueing Network when the DI Layers are in the Middle



The global queueing model is the collection of two queueing networks, one is obtained from the $DTRG_{hdi,ldi}$ for the DI layers, and the other is the generic queueing model for the II layers. These two networks are solved together using an iterative algorithm.

5.5 Iterative Solution of the Queueing Network

In order to solve the queueing network combining networks for the DI and II layers, the iterative solution algorithm given in [Con 88] is used. Based on this solution technique, a network for the DI layers is added to the partitioning of the queueing network for the II layers, as shown in Fig. 5.9. The iterative algorithm given in [Con 88] starts with the initialization of the mean performance measures, i.e. the throughput, mean queue-length, and mean waiting-time of each class of customers of all chains at each queue for every reduced network.

Assume that the number of queues is N, the number of chains is R, the mean service-time for a customer of chain r at queue i in class c is $m_{ir}(c)$, the number of servers at queue i is M_i, the population of chain r is K_r, the population vector is $\mathbf{K} = (K_1, \ldots, K_R)$, and the visit-ratio for class c customers of chain r at queue i is $\alpha_{ic}(r)$. The initial mean performance measures, i.e. the throughput, mean queue-length, and the mean waiting-time of class c customers of chain r at queue i are denoted by $T_{ir}^{(c)}(\mathbf{K})$, $Q_{ir}^{(c)}(\mathbf{K})$, and $W_{ir}^{(c)}(\mathbf{K})$, respectively. Similarly, $T_{ir}(\mathbf{K})$, $Q_{ir}(\mathbf{K})$, and $W_{ir}(\mathbf{K})$ denote the same figures for each chain, and $Q_i(\mathbf{K})$ denotes the mean-queue length of queue i.

The initial mean performance figures are given as

$$Q_{ir}(\mathbf{K}) = e_{ir} t_{ir} K_r / \sum_{i=1}^{N} e_{ir} t_{ir}$$

$$T_{ir}(\mathbf{K}) = e_{ir} K_r / \left[\sum_{i \in I_1} e_{ir} t_{ir} [1 + (Q_i(\mathbf{K}) - Q_{ir}(\mathbf{K})/K_r)] + \sum_{i \in I_2} e_{ir} t_{ir} [1 + (Q_i(\mathbf{K}) - Q_{ir}(\mathbf{K})/K_r)/M_i] + \sum_{i \in I_2} e_{ir} t_{ir} \right]$$

where I_1 is the set of single server queues, I_2 is the set of multi-server queues, and I_3 is the set of IS queues,

$$e_{ir} = \sum_{c \in C_i(r)} \alpha_{ic}(r) \qquad t_{ir} = \sum_{c \in C_i(r)} \alpha_{ic}(r) m_{ir}(c)/e_{ir} \qquad Q_i(\mathbf{K}) = \sum_{r=1}^{R} Q_{ir}(\mathbf{K})$$

and $C_i(r)$ is the set of class of chain r at queue i. The initial mean performance figures for each class of jobs is given as

$$T_{ir}^{(c)}(\mathbf{K}) = \alpha_{ic}(r) T_{ir}(\mathbf{K})/e_{ir} \qquad Q_{ir}^{(c)}(\mathbf{K}) = \alpha_{ic}(r) Q_{ir}(\mathbf{K})/e_{ir}$$

The iteration step of the algorithm is based on solving the reduced networks at each depth in succession while updating the service-time requirements at the flow-equivalent queues. This procedure is repeated until convergence is obtained based on the criteria

$$\left| Q_{ir}(\mathbf{K})^{(j)} - Q_{ir}(\mathbf{K})^{(j)} \right| / K_r < \epsilon$$

where j is the iteration index and ϵ is the desired tolerance.

Once the queueing network is solved, then the measure for the effectiveness of the special-purpose component, e.g. PDU ED can be computed.

5.6 Computation of Effectiveness

The solution of the queueing network for the DI and II layers give the mean performance figures associated with the transfer of every PDU encountered in the communication. In order to obtain the rate of data transfer per unit connection time between the users of two applications, the mean delay time to establish the connection, to release the connection, as well as the delay associated with the data transfer should be computed to obtain the mean connection time for the transfer of a given size of data. The unit of data size is based on the type of application. When the application is file transfer, the unit of data size should be the number of bytes of data files transferred

from one user to another. On the other hand, when the application is a directory service, a more appropriate unit is the number of queries made during a connection. Since all applications use PDUs for communication, a more appropriate measure is the number of PDUs transferred.

The total connection time T_{con} to transfer M identical size data PDUs is the sum of delays associated with the connection establishment T_{ce}, connection release T_{cr} which are independent of the amount of data and the delay associated with the data transfer phase MT_d where T_d is the delay associated with the transfer of one data PDU. Assuming that there exists no connections between the entities at the layers below the top layer, T_{ce} includes the delays associated with the connection establishment at all layers. Similarly, delays associated with data transfer and connection release comprise of the delays encountered at every layer.

In order to evaluate the effectiveness of a special-purpose component of a layered communication architecture, the end-to-end performance of the communication architecture should be measured when the special-purpose component is present and is not used. When such a component is not used, the processing overhead associated with the component is put onto other existing resources of the communication architecture. Then the speed-up in the end-to-end performance should be measured against the amount of resources in the special-purpose component.

When evaluating the effectiveness of PDU EDs, the amount of resources is chosen to be the total number of execution units in IMPs of the architecture. Processors for CC and IC are not added to this count since, they are used to coordinate the computation and data transfers; and regardless of the number of execution units, they are present in IMPs. The following notations are used in the performance analysis:

- k : total number of EUs in a PDU ED,

• $T_{con}, T_{ce}, T_{cr}, T_d$: the total connection time, the connection establishment delay, the connection release delay, and the data transfer delay for a unit PDU when a PDU ED is not used,

• $T^*_{con}, T^*_{ce}, T^*_{cr}, T^*_d$: the total connection time, the connection establishment delay, the connection release delay, and the data transfer delay for a unit PDU when a PDU ED is used,

• S_k : the speed-up of a layered communication architecture using a PDU ED with k EUs over a layered communication architecture using other resources for PDU encoding and decoding,

• η : the effectiveness of the PDU ED.

The total connection time to transfer M unit size data PDUs based on whether or not the PDU ED is used are

$$T^*_{con} = T^*_{ce} + MT^*_d + T^*_{cr}$$

$$T_{con} = T_{ce} + MT_d + T_{cr}$$

The speed-up obtained when the PDU ED is used is

$$S_k = \frac{T^*_{con}}{T_{con}} = \frac{T^*_{ce} + MT^*_d + T^*_{cr}}{T_{ce} + MT_d + T_{cr}} = \frac{(T^*_{ce} + T^*_{cr})/M + T^*_d}{(T_{ce} + T_{cr})/M + T_d}$$

The effectiveness of the PDU ED is

$$\eta = \frac{S_k}{k} = \frac{(T^*_{ce} + T^*_{cr})/M + T^*_d}{k\left[(T_{ce} + T_{cr})/M + T_d\right]}$$

When the number of data PDUs is very large, the effect of the connection establishment and the connection release delays on the total connection time becomes negligible. In this case, the limiting speed-up and the limiting effectiveness become :

$$\lim_{M\to\infty} S_k = \frac{T^*_d}{T_d} \quad , \quad \lim_{M\to\infty} \eta = \frac{T^*_d}{kT_d}$$

Having described the various concepts related with PDU encoding / decoding, chapter 6 will attempt to put these concepts into perspective through an application.

Chapter 6 AN APPLICATION : ASN.1 ENCODING AND DECODING

In this chapter, an application which exemplifies all the items discussed in the previous chapters is given.

6.1 ASN.1

In order to introduce ASN.1, first the concepts of abstract and transfer syntaxes have to be explained. Then, introductions to both the notation and different encoding rules are given in the subsequent subsections.

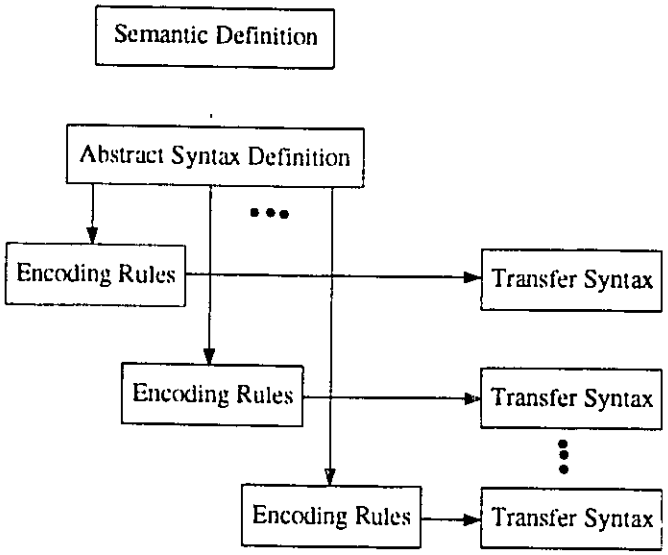
6.1.1 Abstract and Transfer Syntaxes

In the OSI reference model, the nature of the data exchanged between two entities changes when the session layer-presentation layer boundary is crossed. Unlike the lower layers, presentation and application PDUs cannot be specified in an informal way using the help of illustrations. These PDUs which are instances of rather complex data structures necessitate a more formal method. Due to this necessity, ASN.1 is proposed to describe the semantics of PDUs independent of the particular environment as well as to provide a basis to construct a canonical representation which can be used as the global value of these PDUs.

The collection of definitions of PDUs of an Application Service Element (ASE) or the presentation layer is called the *abstract syntax* definition. The possible set of global values for an abstract syntax according to a particular set of encoding rules is called a *transfer syntax* for the abstract syntax as shown in Fig. 6.1. Together, the abstract

and transfer syntax constitute the *presentation context* which is the agreement of two presentation layer entities for a connection. The transfer syntax negotiation and ASN.1 encoding / decoding is conceptually associated with the presentation layer; nevertheless, an implementation may use an ASN.1 ED which provides service to both the application and presentation layer entities.

Figure 6.1 Relations between Syntaxes



6.1.2 Introduction to ASN.1

Similar to type-value duality for PDUs, type and value are basic concepts of ASN.1. An ASN.1 value is an instance of a particular data structure or information item, whereas an ASN.1 type is a set of one or more values.

As all data specification languages, ASN.1 provides a number of *built-in types*, as well as a number of tools with which *constructed types* can be defined. There are two kinds of tools, type and subtype constructors. A *type constructor* is used to define types which include values of other types. A *subtype constructor* is used to define types which

include only a subset of the values of another parent type. The built-in types and the constructor tools of ASN.1 are given in Table 6.1.

Table 6.1 ASN.1 Built-in Types and Constructor Tools

Built-in Types		**Type**	**Subtype**
Simple Types	*Useful Types*	**Constructors**	**Constructors**
BOOLEAN	TIME	ENUMERATED	SINGLE VALUE
INTEGER	EXTERNAL	SEQUENCE	CONTAINED SUBTYPE
BIT STRING	OBJECT DESCRIPTOR	SEQUENCE OF	VALUE RANGE
OCTET STRING		SET	SIZE CONSTRAINT
NULL		SET OF	PERMITTED ALPHABET
OBJECT IDENTIFIER		CHOICE	INNER SUBTYPING
REAL		TAGGED	
CHARACTER STRING			
ANY			

The built-in types of ASN.1 are classified as *simple* built-in types and *useful* built-in types. The simple built-in types are notationally integral to ASN.1. Some of the simple built-in types, e.g. *Boolean*, *Integer*, *Real* are similar to those found in any programming language. The *Bit String* type and the *Octet String* type are variable length strings of bits and bytes, respectively. Other simple built-in types are the *Null* type which comprises of a single value *Null*, the *Object Identifier* type which is used to name standard and user-defined information object classes, various forms of *Character String* types which comprise of an ordered sequence of a variable number of characters, and the *Any* type which can be considered as the union of all types. As an extension of Any type, the construct *Any Defined By* is used when the type of the value of Any is selected by another type's value present in the abstract syntax. The useful built-in types are defined by type constructors. The useful built-in types are the *Time* type which is used to identify points in

time, the *External* type which constitutes the instances of information object classes, and the *Object Descriptor* type which is used to show the textual descriptions of those classes.

The *Enumerated* type constructor is used to define named integers. The *Sequence* type constructor is similar to records in programming languages and is used to define types whose values are ordered collections of values of its elements. The Sequence type constructor is used to specify an ordered set of zero or more element types which need not all be distinct. When one or more element types are declared to be *optional*, the values of these types need not to be among the values of the type constructed by the Sequence type constructor. One or more consecutive types may be defined optional if and only if their tags and the tag of the first consecutive non-optional element are distinct. An optional element may be associated with a *default* value. The *Set* type constructor is similar to the Sequence type constructor, except that the values of its members are unordered. Therefore all members of the type constructed by the Set type constructor have distinct tags. The *Sequence Of* and the *Set Of* type constructors are used to define types whose values are collections of homogeneous values similar to arrays in programming languages. The *Choice* type constructor is used to define a type that is the union of one or more alternative types with distinct tags similar to variant records of programming languages. The *Tagged* type constructor is used to define a type which differs from a given subject type only by its tag.

The *Single Value* subtype constructor is used to define a type having a single value. The *Contained Subtype* constructor is used to nominate a specified subtype. The *Value Range* subtype constructor is used with parent types, which are either Integer or Real, to specify an interval of values. The *Size Constraint* subtype constructor is used with parent types, which are either Bit String, Octet String, Character String, Sequence Of or Set Of type, to specify values constrained to certain lenghts. The *Permitted Alphabet* subtype constructor is used with parent types, which are of type Character String, to

specify values constructed of only a set of characters. The *Inner Subtyping* constructor has different forms for different parent types. If the parent type is Sequence Of or Set Of types, the Inner Subtyping constructor specifies the type obtained from the parent type by constraining its element / member type. When the parent type is Sequence or Set type, the Inner Subtyping constructor specifies the type obtained from the parent type by constraining one or more element / member types. If the parent type is Choice type, the Inner Subtyping specifies the type obtained from the parent type by excluding one or more of alternative types.

Each ASN.1 type is associated with a descriptor called a *tag* which uniquely identifies that type. Tags have the same functionality as the identifiers of PDU types. Similarly, the encoding rules guarantee that tags for ASN.1 types appear in the global values, such that values of one type can be distinguished from values of another type. ASN.1 built-in types and types defined by type constructors have tags. Based on the tagging mechanism, the Tagged type constructor is used to identify types uniquely and unambiguosly.

A tag consists of a class and a number. The *class* of a tag specifies the domain of the tag's *number* which is a non-negative integer. There are four classes of tags :

- *Universal* tags are assigned within the ASN.1 standards. A distinct tag is assigned to each built-in type, except Any, and each type constructor except Choice and Tagged.
- *Application* tags are assigned within each ASN.1 module.
- *Context-specific* tags are assigned within each context, typically formed by the alternative types of a Choice type, the element types of a Sequence type, or the member types of a Set type.
- *Private* tags are assigned within each organization, and are used for user-defined types.

When the Tagged type constructor is used, it is possible to choose two modes of tagging, *explicit* and *implicit*. The tagging mode defines the way of encoding for types

defined by the Tagged type constructor.

All the constructs including the built-in types and their values, as well as the types constructed by the tools and their values can be used in definitions for application and presentation PDUs. The type definitions for related data are collected into *modules* which are uniquely identified by Object Identifier values. All types and values in a given module can be made accesible by using the notational constructs, namely *importing* and *exporting*. The default tagging mode can also be set for each module.

Fig. 6.2 shows an example ASN.1 module to specify a PDU type, named *Exp-PDU* which is defined in terms of *TypeA*, which is itself defined by *TypeB* and *TypeC*. According to the definitions, the values of *Exp-PDU* are sequences consisting of an arbitrary number of values of *TypeA*. The values of *TypeA* optionally have the values of *TypeB* as their first component, and always have the values of *TypeC*. *TypeB* is obtained by the application of the Size Constraint subtype constructor on the type *Set Of Integer* such that its values always have two member values. Similarly, the values of *TypeC* always have the value of a specific Character String as their first element, and optionally, the value of an Integer as the second element.

Figure 6.2 An Example ASN.1 Module

```
EXAMPLE DEFINITIONS ::=
BEGIN
    Exp-PDU ::= [APPLICATION 0] IMPLICIT SEQUENCE OF TypeA
    TypeA ::= SEQUENCE {first    [0] IMPLICIT TypeB OPTIONAL,
                        second  [1] IMPLICIT TypeC}
    TypeB ::= SET SIZE (2) OF INTEGER
    TypeC ::= SEQUENCE {s1  IA5String,
                        s2  INTEGER DEFAULT 1}
END
```

The ASN.1 notation allows its user to extend itself with defining alternative, and non-standard notation for referencing types and values by using *macros*. A macro definition provides distinctive type and value notation, and it uses this extended notation to define new types or values.

A macro definition consists of two grammars of production rules. A macro use which denotes a type consists of a sentence that can be produced from the TYPE NOTATION production rules. A macro use which denotes a value consists of a sentence that can be produced from the VALUE NOTATION. As an example, Fig. 6.3 shows the definition for the OSI Remote Operations (ROS) [ISO 9072–1] OPERATION macro. ROS is the OSI's paradigm for remote operations requested by one open system to be carried out at another. The macro's subject type is a Choice type whose alternative types are Integer and Object Identifier. This non-standard notation is used to define a class of operations which are performed at one open system by the request of another. Each operation is identified by the specified Integer or Object Identifier value associated with it. Therefore, by simply including the value for an Integer type or an Object Identifier type in a global PDU value, a whole set of semantics related with the particular operation can be sent from one open system to another.

Figure 6.3 OPERATION Macro for OSI Remote Operations

```
OPERATION MACRO ::=
BEGIN
TYPE NOTATION  ::= Argument Result Errors LinkedOperations
VALUE NOTATION ::= value(VALUE CHOICE {
                                              localValue     INTEGER,
                                              globalValue    OBJECT IDENTIFIER})
ARGUMENT            ::= "ARGUMENT" NamedType | empty
Result              ::= "RESULT" ResultType | empty
ResultType          ::= NamedType | empty
Errors              ::= "ERRORS" "{" ErrorNames "}" | empty
LinkedOperations    ::= "LINKED" "{" LinkedOperationNames "}" | empty
ErrorNames          ::= ErrorList | empty
ErrorList           ::= Error | ErrorList "," Error
Error               ::= value(ERROR) | type
LinkedOperationNames ::= OperationList | empty
OperationList       ::= Operation | OperationList "," Operation
Operation           ::= value(Operation) | type
NamedType           ::= identifier type | type
END
```

The ROS specifications include six macros, namely, BIND, UNBIND, OPERATION, ERROR, APPLICATION-SERVICE-ELEMENT and APPLICATION-CONTEXT. These macros are used to define types in abstract syntaxes for OSI applications using ROS. As an example, Fig. 6.4 shows the definition for Remote Operations Service Element (ROSE) Remote Operations Invoke PDU (*ROIVapdu*). The implementation details about the use of ROS macros in other OSI applications are explained in subsection 6.3.2.

Figure 6.4 ROSE ROIVapdu Definition

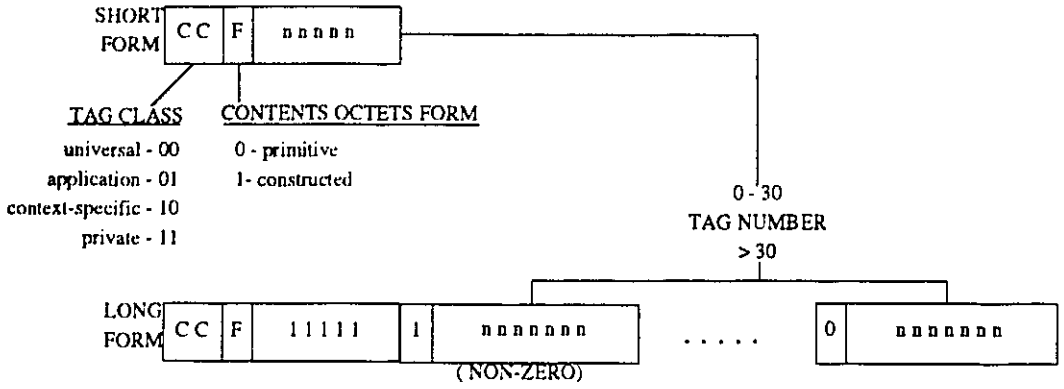
```
Remote-Operations-APDUs DEFINITIONS ::=
BEGIN
.....
ROSEapdus          ::= CHOICE {
                           roiv-apdu [1] IMPLICIT ROIVapdu,
                           rors-apdu [2] IMPLICIT RORSapdu,
                           roer-apdu [3] IMPLICIT ROERapdu,
                           rorj-apdu [4] IMPLICIT RORJapdu}

ROIVapdu           ::= SEQUENCE {
                             invokeID        InvokeIDType,
                             linked-ID [0]   IMPLICIT InvokeIDType OPTIONAL,
                             operation-value OPERATION,
                             argument        ANY DEFINED BY operation-value OPTIONAL}

InvokeIDType       ::= INTEGER
.....
END
```

6.1.3 Introduction to BER

The Basic Encoding Rules (BER) are currently the only standard encoding rules for ASN.1. According to the BER, every component instance in a global value consists of three functional units; the identifier, the length, and the contents. In other words, according to the BER no functional unit is implicit except the case when the coding of the length unit is zero, in which case there is no contents unit.

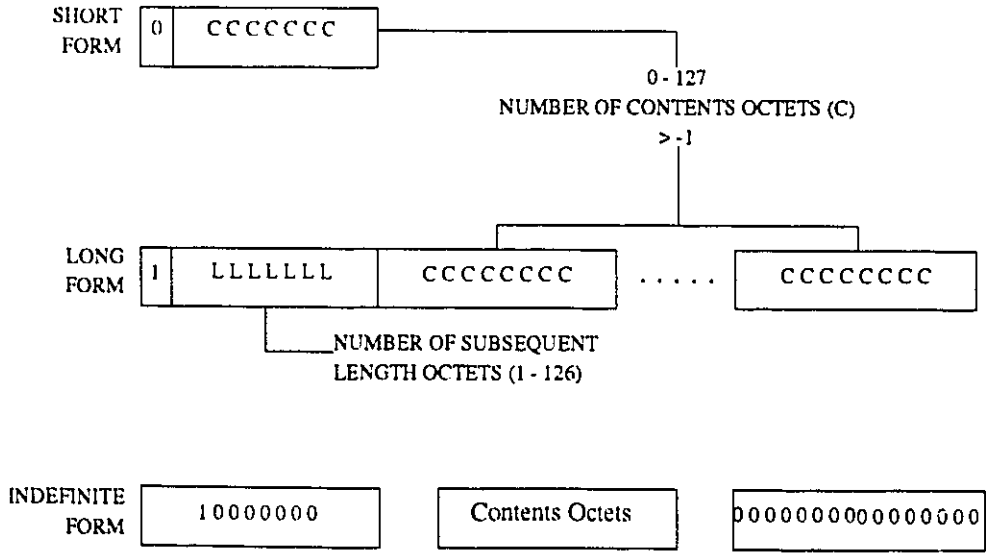
The identifier unit encodes the value's tag and the encoding form. It takes either the *short* form comprising of a single octet for types with tag numbers up to 30, or the *long* form comprising of two or more octets for types with greater tag numbers as shown in Fig. 6.5.

Figure 6.5 Identifier Unit According to BER [Whi 89]



The length unit is either definite or indefinite. The definite length unit can be in either the *short* form consisting of one octet and used to encode up to 127, or the *long* form consisting of two or more octets which encode up to $2^{1008}-1$. When the indefinite form which can only be used for constructed encodings is used, the contents unit is enclaved between an indefinite marker consisting of one octet and an indefinite trailer of two octets. Different forms of length unit are shown in Fig. 6.6.

Figure 6.6 Length Unit According to BER [Whi 89]



The contents unit takes either the primitive or the constructed form. The primitive form consists of zero or more octets which are collectively the encoding of the single value. The constructed form consists of the encodings of zero or more other values whose meaning depends on the type of the value being encoded.

According to the BER, the contents unit of values of Boolean, Integer, Real, Object Identifier, Null, and Enumerated types takes only the primitive form, whereas the contents unit of values of Sequence, Sequence Of, Set, Set Of, and External types takes only the constructed form.

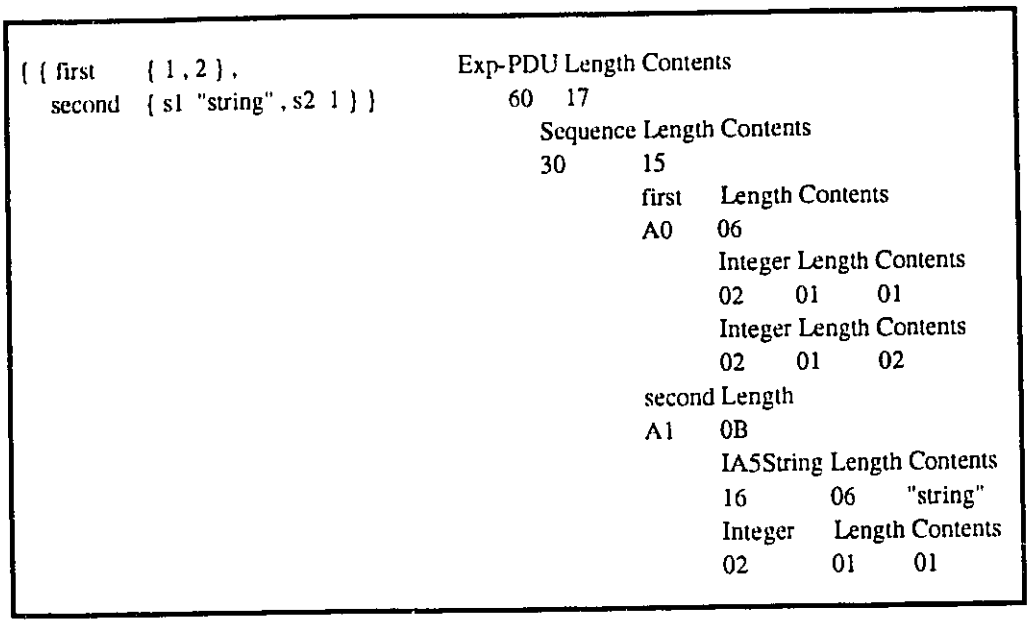
According to the BER, both the primitive and the constructed form can be used for the contents units of values of Bit String, Octet String, Character String, Time, and Object Descriptor types. When the contents units of encodings of strings are in the constructed form, they are composed of encodings of zero or more string values, each representing a portion of the entire string. The contents units of string values representing portions may also be in the constructed form.

The contents units of the values of the Tagged type may also take either the primitive or the constructed form. If the tagging mode is explicit, the contents unit is in the constructed form and consists of the entire encoding of the value of the subject type. If the tagging mode is implicit, the contents unit is in either the primitive or the constructed form – depending on the subject type – and consists of only the contents unit of the encoding of the value of the subject type.

The encoding choice for a value of the Any type, and the Choice type depends on its subject type, and the selected alternative type, respectively.

An example value for *Exp-PDU* of Fig. 6.2, and its transfer syntax according to the BER are given in Fig. 6.7.

Figure 6.7 A Value of Exp-Pdu and Its Transfer Syntax According to the BER



Lemma 6.1 : ASN.1 decoding based on the BER is fully divisible.

Proof : A global PDU value $g = f_1 \ldots f_n$ encoded according to the BER always starts with the identifier unit for the root node of the corresponding local value. According to the BER, no functional unit is implicit and functional units for a local value node are always in the same order : identifier, length, and contents, regardless of the type. Therefore, while processing from left to right,
$\forall f_i\ 1 < i \leq n \exists f_j \bullet (1 \leq j < i) \wedge \mathbf{P}_{BER}(i) \prec \mathbf{P}_{BER}(j)$. []

6.1.4 Other Encoding Rules

Different encoding rules other than the BER were also proposed for different purposes. We examine two of these which are currently under review to become international standards.

In the OSI reference model, some applications require the generation and verification of digital signatures computed from global PDU values. These signatures are used for

authentication and security purposes. According to the BER, the global value for an ASN.1 value is not unique due to the encoding choices such as using either primitive or constructed form for the contents unit or different forms of the length unit. Since the BER do not generate unique encodings, the encodings cannot be used as the basis of generating digital signatures. To overcome this difficulty, the *Distinguished Encoding Rules (DER)* [ISO 8825–3] are proposed. According to the DER, a unique global PDU value is generated for each local PDU value. Some example restrictions are : using only the primitive form for the contents unit of strings, using only the indefinite length format, or specifying a certain order for the encodings of the members of a set value.

Corollary : ASN.1 decoding based on the DER is fully divisible.

Proof : Follows from $G_{P,DER} \subset G_{P,BER}$ for any protocol P. []

One of the major criticisms about the BER is the level of verbosity in the encodings. Since the identifier, and the length units are always present for every local value node, there is a possibility of redundancy when these units are implicitly given in the functional units of other nodes. In order to minimize the number of octets in the global values, the *Packed Encoding Rules (PER)* [ISO 8825–2] are proposed. According to the PER, the identifier and length units for a local value node are not included in the global value whenever they can be deduced from other functional units. Therefore, according to the PER a functional unit may be implicit. However, since this implicit unit cannot be gathered without knowing the corresponding type, ASN.1 decoding based on the PER is not divisible. Therefore, no parsing takes place and over type-value matching for decoding becomes the only phase for ASN.1 decoding according to the PER.

6.2 Software-based Implementations

Several software tools such as compilers [Neu 90], libraries inside communication

packages [Wol 88], and translators [Bra 90], have been developed to support ASN.1. Aside from the translators which convert a given ASN.1 definition into a definition in another data specification language, e.g. ACT One, all tools include ASN.1 encoding and decoding based on the BER. In this section, different models for software-based ASN.1 EDs are discussed.

6.2.1 Encoding / Decoding with an Intermediate Form

ASN.1 encoding / decoding using an intermediate form for the representation of PDUs is the approach taken in the ISO Development Environment (ISODE). The ISODE is a software package developed by the Wollongong Group and the Northrop Corporation for OSI-based applications [Wol 88]. As part of its structure, the ISODE contains a set of ASN.1 tools and libraries. The main design objective of this project is to provide ASN.1 encoding / decoding based on the BER to different ASEs existing in the ISODE.

The approach of the ISODE for ASN.1 encoding / decoding is similar to the PDU encoding / decoding discussed in chapter 2. ASN.1 PDU values are represented in a format called the *presentation element* which is similar to the intermediate value form. Another format called the *presentation stream* is used to represent the encoded values similar to the global value form. There are library routines, which provide the mapping between the presentation elements and presentation streams, corresponding to parsing and assembling transformations.

The routines, which provide the mapping between the presentation elements and the local values of the PDUs, are generated from the ASN.1 definitions by an ASN.1 compiler. The compiler also generates the C data structures for representing the local values.

A similar approach is taken in the National Bureau of Standards (NBS) ASN.1 tools based on an object-oriented model [Gau 89]. In this model, object-oriented definitions

for ASN.1 types, and values are used. The object-oriented technique is proposed as a way of abstraction of the concepts of ASN.1. The model is used for a set of ASN.1 tools by using object-oriented programming techniques and syntax-directed translation.

According to the design of NBS ASN.1 tools, translation and activation are phases of the design problem. Translation is used to convert an abstract syntax into an intermediate representation which includes all the infomation in a format suitable for different computational requirements. Activation is used to add needed procedures to the output of the translator. As an example, ASN.1 encoding / decoding tool is obtained from ASN.1 encoding / decoding activation.

In the object-oriented model, the *TypeDesc* (type descriptor) class and the *ValueDesc* (value descriptor) class are used for specifying type and value definitions, respectively. ASN.1 types and constructor tools given in Table 6.1, and their respective values are abstracted by subclasses of the *TypeDesc* and the *ValueDesc*. Similar to our model, different forms for values, i.e. transfer value, local value, print value, and filed value are defined.

The encoding / decoding is based on the transformations between the transfer value and the intermediate value structures using the *TypeDesc*. The same intermediate value format can be used to obtain the local values, print values which are given in ASN.1 value notation, and filed values which are the collections of intermediate values in files.

6.2.2 Piecewise Encoding / Decoding

As opposed to ASN.1 encoding / decoding using an intermediate form of representation for PDU values, piecewise encoding / decoding is based on generating the routines for each PDU of a given abstract syntax. CASN1 [Neu 90], an ASN.1–C compiler along

with an implementation of the BER is a typical example of the piecewise encoding / decoding.

According to the approach of CASN1, there is no embedded type information in the encoding / decoding. The compiler provides its user with a set of procedures to encode / decode every item in the abstract syntax. The main goal of the approach is to reduce the memory requirements by eliminating the use of an intermediate form and by building a specialized memory management system.

Similar to the ISODE and the NBS system, CASN1 produces C data structures as well as the encoding / decoding routines for items in the abstract syntax from a given ASN.1 definition.

Now having discussed different software implementations, we can introduce the details of our phase algorithms for ASN.1 encoding / decoding according to the BER and the DER.

6.3 Phase Algorithms for ASN.1 Encoding / Decoding

In this section, we discuss the details of the phase algorithms for ASN.1 encoding and decoding according to the BER and DER. ASN.1 encoding / decoding according to the PER is not addressed due to the fact that ASN.1 decoding according to the PER cannot be divided into phases. Therefore it has to be redefined in a single phase and this would result in a rather complex description.

6.3.1 Parsing Algorithm for ASN.1 Decoding

In Lemma 6.1 and its corollary, it is shown that ASN.1 decoding according to BER or DER is fully-divisible. Therefore, the global PDU values, i.e. encoded ASN.1 values

can be separated into functional units and using these functional units an intermediate value can be obtained.

Intermediate values for ASN.1 decoding are represented by general trees whose nodes have attributes for the identifier, length, and contents units. In Fig. 6.8, the C data structure for an intermediate value node is shown. According to the data structure, intermediate value nodes are linked to each other by the leftmost child and the right sibling pointers. The attribute for the contents unit is a pointer to the beginning of the string of contents octets. The contents unit pointer is set to *nil* when the node is for a value of a constructed type. Since, only the definite length encoding is used for the values of primitive types, the contents pointer and the length attribute is sufficient to access the contents octets. If a node does not have a child and/or a right sibling, the respective pointers are also set to *nil*.

Figure 6.8 C Data Structure for Intermediate Value Node for Decoding

```
typedef unsigned char byte;
typedef struct IVNd {
        int             identifier;         /* identifier unit for the intermediate value node */
        int             length;             /* length unit for the intermediate value node */
        byte*           *contents;          /* contents unit for the intermediate value node */
        struct IVNd     *leftmost_child;    /* pointer to the leftmost child intermediate value node */
        struct IVNd     *right_sibling;     /* pointer to the right sibling intermediate value node */
}       IVNd, *ptrIVNd;
```

Now, based on the data structure specified for intermediate values we can explain the details of ASN.1 parsing.

According to definition 2.4.2, for a given global value $g = f_1 \ldots f_M$, there are precedence relations between the parsing transformations of individual functional units. Parsing of a global value encoded according to the BER or DER starts with the transformation of the first functional unit f_1. The precedence relations between different

functional units for any BER or DER encoded global value is given in Table 6.2 where $\mathcal{N}_k(f_i)$, and $\mathcal{F}_k(f_i)$ denote the intermediate value node, and the functionality for unit f_i computed at the k^{th} step of the algorithm, respectively. The set of functional units whose parsing transformations are preceded by that of f_i is denoted by $S_{k,i}$. The left child and the right sibling of an intermediate value node in_j is denoted by $lc(in_j)$, and $rs(in_j)$, respectively.

Table 6.2 Precedence Relations between Functional Units for ASN.1 Parsing Algorithm

$\mathcal{N}_k(f_i)$	$\mathcal{F}_k(f_i)$	$S_{k,i}$
in_j	*identifier*	$\{f_{i+1}\}\bullet\mathcal{N}_{k+1}(f_{i+1})=in_j\wedge\mathcal{F}_{k+1}(f_{i+1})=length$
in_j	*length*	$(in_j\ primitive\wedge in_j.length\neq 0)\Rightarrow\{f_{i+1}\}\bullet\mathcal{N}_{k+1}(f_{i+1})=in_j\wedge\mathcal{F}_{k+1}(f_{i+1})=contents$
		$(in_j\ primitive\wedge in_j.length=0\wedge\neg(\exists l\leq k\bullet\mathcal{N}_l(f_{i+1})=rs(in_i)\wedge in_j\ child\ of\ in_i))\Rightarrow$
		$\mathcal{N}_{k+1}(f_{i+1})=rs(in_j)\wedge\mathcal{F}_{k+1}(f_{i+1})=identifier$
		$(in_j\ constructed\wedge in_j.length\ definite\wedge in_j.length\neq 0)\Rightarrow\{f_{i+1},f_m\}\bullet$
		$\mathcal{N}_{k+1}(f_{i+1})=lc(in_j)\wedge\mathcal{F}_{k+1}(f_{i+1})=identifier\wedge$
		$\mathcal{N}_{k+1}(f_m)=rs(in_j)\wedge\mathcal{F}_{k+1}(f_m)=identifier\wedge\lvert f_{i+1}\ldots f_{m-1}\rvert=in_j.length$
		$(in_j\ constructed\wedge in_j.length\ definite\wedge in_j.length=0)\Rightarrow\{f_{i+1}\}\bullet$
		$\mathcal{N}_{k+1}(f_{i+1})=rs(in_j)\wedge\mathcal{F}_{k+1}(f_{i+1})=identifier$
		$(in_j\ constructed\wedge in_j.length\ indefinite)\Rightarrow\{f_{i+1}\}\bullet$
		$\mathcal{N}_{k+1}(f_{i+1})=lc(in_j)\wedge\mathcal{F}_{k+1}(f_{i+1})=identifier$
in_j	*contents*	$\neg(\exists l\leq k\bullet\mathcal{N}_l(f_{i+1})=rs(in_i)\wedge in_j\ child\ of\ in_i)\Rightarrow\{f_{i+1}\}\bullet$
		$\mathcal{N}_{k+1}(f_{i+1})=rs(in_j)\wedge\mathcal{F}_{k+1}(f_{i+1})=identifier$

The algorithm for the parsing of a given global value encoded according to the BER or DER is an instance of algorithm 2.1 based on the precedence relations given in Table 6.2. The only issue which is not trivial while addressing the parsing of global values

encoded according to BER is the presence of length units in indefinite format. Since all length units are in definite format according to the DER, the precedence relations in Table 6.2 are sufficient for the parsing algorithm.

When the length unit for an intermediate value node is in indefinite format, the parsing algorithm should find the matching end-of-contents marker for the node. In order to find the matching pairs of an indefinite length unit and an end-of-contents marker, the parsing algorithm uses a stack. Each time an indefinite length unit is found, the intermediate value node using this length unit is put into a stack. When an end-of-contents marker is found it marks the end of contents for the intermediate value node at the top of the stack. Obviously, the indefinite length units and end-of-contents markers in a given global value must be balanced.

Because of the use of indefinite length units for values of constructed types, it is quite cumbersome to use the length of functional units to determine the rightmost child of an intermediate value node whose length unit is definite. Instead, as specified in the last two lines of Table 6.2, after processing a contents unit the next functional unit is checked, whether or not it is processed at a previous step. If the next functional unit is processed at a previous step, the intermediate value node for the contents unit is the rightmost child of its parent. In order to store the processed functional units, a table, whose entries are functional unit addresses, is used.

Now having explained the details of the ASN.1 parsing algorithm implementation, we can introduce the type-value matching for decoding algorithm.

6.3.2 Type-Value Matching Algorithm for ASN.1 Decoding

Since ASN.1 decoding according to the BER and DER is shown to be fully-divisible, the type-value matching for decoding transformation is also complete as the parsing

transformation. In type-value matching for decoding transformation, the intermediate value is converted into a local value based on the type tree of the current abstract syntax.

Type trees for type-value matching, for both encoding and decoding algorithms, are represented by general trees whose nodes have attributes for the identifiers of types, and implicit types, flags to mark types as default, optional, and a pointer to a complementary information table where additional information about certain types is stored. Type trees and the complementary information table for each abstract syntax are obtained from its ASN.1 definitions using a tool developed from the CASN1 compiler introduced in the previous section.

In Fig. 6.9, the C data structure for a type tree node, which corresponds to an ASN.1 type other than Choice type, is shown. Similar to intermediate value nodes, type tree nodes are connected to each other by the leftmost child and the right sibling pointers. However, as explained below, since there is more than one type of node in the type tree, the leftmost child and right sibling pointers are defined as a collection of three node types. The *identifier* attribute for the type tree node consists of the class, expected encoding form and the tag number for the corresponding ASN.1 type. The *flags* attribute of a type tree node contains flags to show whether or not the type is declared as Optional, Default, or is a homogeneous collection, i.e. Sequence Of or Set Of type. The *implicit* attribute of a type tree node contains the implicit Universal type of an implicit Tagged type. The *definednode* attribute is a pointer to a DynamicChoice type used to handle the use of macro in ROSE PDUs and it is explained later in detail. The *complementary* attribute of a type value node points to the type node's complementary table entry which is generated for types with enumerated values, types with default values, and constraints for subtypes.

Figure 6.9 C Data Structure for a Type Tree Node

```
typedef unsigned char byte;
typedef struct  TTN {
        int             identifier;         /* identifier for the type tree node */
        byte            flags;              /* flags for the type tree node */
        byte            implicit;           /* implicit identifier for the type tree node */
        struct DCN      *definednode;       /* pointer to the referenced DynamicChoice type node */
        byte*           *complementary;     /* pointer to the complementary information table */
        union {
                struct TTN    *next_tn;     /* leftmost child is standard type tree node */
                struct CTN    *next_cn;     /* leftmost child is choice type node */
                struct DCN    *next_dcn;    /* leftmost child is dynamic choice node */
                }       *leftmost_child;
        union {
                struct TTN    *next_tn;     /* right sibling is standard type tree node */
                struct CTN    *next_cn;     /* right sibling is choice type node */
                struct DCN    *next_dcn;    /* right sibling is dynamic choice node */
                }       *right_sibling;
}       TTN, *ptrTTN;
```

The type tree node structure shown in Fig. 6.9 has only two pointers to link it to other type tree nodes. However, if a type is defined to be a Choice type, this structure becomes rather inefficient, since in order to find the alternative type for the corresponding value, the type tree nodes for these alternative types need to be accessed sequentially. In order to overcome this difficulty, type tree nodes for Choice types are augmented with an arbitrary number of pointers to its children, i.e. the alternative types. Each alternative type pointer is complemented with a unique identifier which is the identifier unit for an instance of the alternative type. Since the number of children is arbitrary, an attribute encoding this number is also used in the type tree nodes for Choice types. In Fig. 6.10, the C data structure for a type tree node, which corresponds to a Choice type, is shown.

A problem which needs to be solved while generating type trees is the embedded Choice types. If an alternative type is also defined to be a Choice type, then it has to be expanded to generate the pairs of unique identifier and address for all the alternative types. Therefore, generated type trees use such flattened out form of embedded Choice types.

Figure 6.10 C Data Structure for a Choice Type Tree Node

```
typedef unsigned char byte;
typedef struct  id&adr {
                        int             identifier;     /* unique  identifier for the alternative type */
                        struct TTN      *address;       /* pointer to the alternative type */
                        }       id&adr;
typedef struct  CTN {
        int             identifier;                     /* identifier for the choice type tree node */
                                                        /* if it is not a Tagged type identifier is set  to 0 */
        byte            flags;                          /* flags for the choice type tree node */
        byte            no_of_alternatives;             /* number of alternatives for the choice type  < 256 */
        struct id&adr   alternatives[255];              /* alternative types */
        byte*           *complementary;                 /* pointer to the complementary information table */
        union   {
                struct TTN      *next_tn;               /* right sibling is standard type tree node */
                struct CTN      *next_cn;               /* right sibling is choice type node */
                struct DCN      *next_dcn;              /* right sibling is dynamic choice node */
                }       *right_sibling;
}       CTN, *ptrCTN;
```

Another special type tree node is generated for the implementation of the ROS macros. In our ASN.1 implementation, the general use of macro notation is not considered. The main reason for this decision is that while the virtually infinite range of non-standard notation requires separate and complex tools, such as a parser for each macro, the expressive power of the notation does not drastically increase. On the other hand, the use of ROS macros is standardized and used in different OSI applications.

The difficulty in processing PDUs based on the ROS macros is due to the missing semantic link between the possible alternative types for an *argument*, a *parameter*, or a *result* and the selecting identifier. In ROSE PDUs, Any Defined By construct is used to link arguments and results to operations, and parameters to errors. However, in the ASN.1 standard, the method of implementation for Any Defined By is not defined. In order to provide the missing link, we define a non-standard ASN.1 type called *DynamicChoice* which is the collection of identifiers and corresponding alternative types. In order to better describe the use of DynamicChoice, let us examine the use of ROIVapdu in Directory

Access Protocol [ISO 9594–5] definitions given in Fig. 6.11. According to the definitions given in Fig. 6.11, the argument of ROIVapdu is defined by the operation-value element of the same PDU and can be any one of the nine arguments defined by the operation-value. This relation between two elements of the ROIVapdu is captured by the use of DynamicChoice. It should be noted that these semantic links between named values and corresponding types are put manually.

Figure 6.11 An Example ROSE PDU Using Dynamic Choice

```
ROIVapdu    ::= SEQUENCE {
                    invokeID            InvokeIDType,
                    linked-ID           [0] IMPLICIT InvokeIDType OPTIONAL,
                    operation-value     Operation,
                     argument           DYNAMIC Argument DEFINED BY
                                        operation-value OPTIONAL }

Argument    ::= DynamicChoice {
                      read              ReadArgument,
                      compare           CompareArgument,
                      abandon           AbandonArgument,
                      list              ListArgument,
                      search            SearchArgument,
                      addEntry          AddEntryArgument,
                      removeEntry       RemoveEntryArgument,
                      modifyEntry       ModifyEntryArgument,
                      modifyRDN         ModifyRDNArgument }
```

In order to represent the DynamicChoice types, another form of type tree node is used as shown in Fig. 6.12. The *identifier* attribute for a DynamicChoice type is always set to zero. The *definingvalue* pointer is the address for the contents octets for the defining value whose type is either Integer or Object Identifier according to ROSE. This pointer is generated while the defining value is processed using the address of the DynamicChoice type which is stored as the *definednode* attribute of the type node for the defining value.

Figure 6.12 C Data Structure for a DynamicChoice Type Tree Node

```
typedef unsigned char byte;
typedef struct  id&adr {
                        int             identifier;     /* unique  identifier for the alternative type */
                        struct TTN      *address;       /* pointer to the alternative type */
                        }       id&adr;
typedef struct  DCN {
        int             identifier;                     /* identifier for the DynamicChoice type tree node */
                                                        /* and is set  to 0 */
        byte            flags;                          /* flags for the DynamicChoice type tree node */
        byte*           *definingvalue;                 /* address for the defining value */
        byte            no_of_alternatives;             /* number of alternatives for the choice type  < 255 */
        struct id&adr   alternatives[255];              /* alternative types */
        byte*           *complementary;                 /* pointer to the complementary information table */
        union  {
                struct TTN      *next_tn;               /* right sibling is standard type tree node */
                struct CTN      *next_cn;               /* right sibling is choice type node */
                struct DCN      *next_dcn;              /* right sibling is dynamic choice node */
                }       *right_sibling;
}       DCN, *ptrDCN;
```

Now, based on the structure of type tree nodes, we can explain the details of type-value matching for ASN.1 decoding.

According to definition 2.4.5, an intermediate value node in_i matches a type tree node tn_j iff there is an equivalency function $eq(.)$ such that $eq(in_i.identifier)=tn_j.identifier$. The equivalency function for the DER is rather simple :

$eq(in_i.identifier)=tn_j.identifier \Leftrightarrow in_i.identifier=tn_j.identifier$, since according to the DER there is only one way of encoding for a given ASN.1 value. On the other hand, according to the BER, the contents unit of some built-in types, e.g. Bit String, Octet String, may be either in the primitive or the constructed form. Therefore for such types, the equality of the class and the tag number is sufficient to make their respective nodes match.

As explained in subsection 2.4.2, during type-value matching, instead of the possible match set T_j, an initial type tree node and the method to reach the remaining nodes in T_j are generated at each step. The set T_j has more than one member only when the elements

of a Sequence type or members of a Set type are being processed. When the elements of a Sequence type are considered the method to reach the remaining nodes in T_j is to proceed until finding the first element which is not Optional or has Default value. In the case of Set, the method is more complex since values for members of a Set type may appear in any order in the global value. In this case, the method to reach the remaining nodes in T_j is to compare the intermediate value node against all members for which a matching intermediate value is not found. In order to accomplish this, there needs to be an extra variable which stores the members, for which a matching intermediate value was previously found. When decoding global values encoded according to the DER, no special provision is necessary for the members of a Set type since according to the DER, there is always a fixed order of values for members in the global values.

The local values generated by the type-value matching algorithm combine the relevant attributes of the matching intermediate value tree and type tree nodes. Since local value nodes correspond to different ASN.1 built-in types, which are used by applications, their structures are different. Nevertheless, all local value nodes have some common attributes such as *identifier*, *flags*, and *complementary table pointer*. Specific attributes are also present for contents units in all types of local value nodes, such as *number of substrings for contents* of Bit String, Octet String, and Character String types or the *number of elements* of Object Identifier types. The use of the attribute *encodingchoice* is explained in the next subsection. Two example local value nodes, an integer, and an Octet String, are shown in Fig. 6.13.

Figure 6.13 C Data Structure for Example Local Value Nodes

```
typedef unsigned char byte;
typedef struct  LNInt {
        int             identifier;         /* identifier for the local value node */
        byte            flags;              /* flags for the local value node */
        byte*           *complementary;     /* pointer to the complementary information table */
        int             *contents;          /* contents for the integer local value node */
        byte            encodingchoice;     /* used in encoding to choose the encoding rule */
        union  {
                struct LNBool    *next_bool;   /* right sibling is a Boolean local value node */
                struct LNInt     *next_int;    /* right sibling is an Integer local value node */
                .....
                }        *right_sibling;
}       LNInt, *ptrLNInt;
.....
typedef struct OctetStr {
                        int         numOctets;      /* number of Octets in the substring */
                        byte*       *contents;      /* pointer to the Octet substring */
                        }     OctetStr;
typedef struct  LNOctStr {
        int             identifier;         /* identifier for the local value node */
        byte            flags;              /* flags for the local value node */
        byte*           *complementary;     /* pointer to the complementary information table */
        byte            numsubstr;          /* number of Octet substrings */
        struct OctetStr substrings[255];    /* Octet substrings */
        byte            encodingchoice;     /* used in encoding to choose the encoding rule */
        union  {
                struct LNBool    *next_bool;   /* right sibling is a Boolean local value node */
                struct LNInt     *next_int;    /* right sibling is an Integer local value node */
                .....
                }        *right_sibling;
}       LNOctStr, *ptrLNOctStr;
```

6.3.3 Type-Value Matching Algorithm for ASN.1 Encoding

In type-value matching for encoding transformation, local values are converted into intermediate values according to the type tree of the current abstract syntax. The attribute *encodingchoice* of local value nodes is used in this algorithm to determine which encoding rule is to be applied to generate the global value when there are more than one encoding choices. The use of multiple encoding rules for the same value is valid only according

to the BER; whereas according to the DER there is always one and only one encoding rule to obtain the global value for a given local value.

The type-value matching for ASN.1 encoding algorithm is the dual of the type-value matching for ASN.1 decoding, except that a bottom-up traversal of generated intermediate value nodes is also necessary to compute the length of values for constructed types. Three additional attributes are used in the intermediate value nodes for ASN.1 encoding. The attribute *lengthformat* specifies whether the length unit of a value for a constructed type is definite or indefinite when the encoding rules set is the BER. When the encoding is done according to the DER, there is no need for this attribute since only the definite format is used for length units. The attributes *numchildren*, and *numresponses* respectively record the total number of children, and the number of children from which length information is received for a value of a constructed type. Another difference of the intermediate value node for encoding with respect to that for decoding is that the attribute *length* is the integer value encoding the total length of encodings for all the children of the intermediate value node. Since different formats for length units can be used, the length unit is generated during the assembling transformation. In Fig. 6.14, the C data structure for an intermediate value node used in encoding is shown.

Figure 6.14 C Data Structure for Intermediate Value Node for Encoding

```
typedef unsigned char byte;
typedef struct IVNe {
        int             identifier;     /* identifier unit for the intermediate value node */
        int             length;         /* length of the contents unit */
        byte            lenghtformat;   /* length is definite or indefinite for constructed types */
        byte*           *contents;      /* contents unit for the intermediate value node */
        byte            numchildren;    /* number of children for values of constructed types */
        byte            numresponses;   /* number of children which sent their length */
        struct IVNe     *parent;        /* pointer to the parent node */
        struct IVNe     *leftmost_child;/* pointer to the leftmost child intermediate value node */
        struct IVNe     *right_sibling; /* pointer to the right sibling intermediate value node */
}       IVNe, *ptrIVNe;
```

6.3.4 Assembling Algorithm for ASN.1 Encoding

The assembling algorithm for generating global values encoded according to the BER or DER is based on the pre-order traversal of the intermediate value tree. In section 2.5.2, the generation of the global value is abstractly explained as the logical OR operation performed on partial global values and generated bit streams for the intermediate value nodes. In the implementation for ASN.1 encoding, when functional units are generated, they are written into the partial global value starting at a given address. In order to include the choice of the format for length units of values for constructed types according to the BER, the attribute *lengthformat* is used to determine the preferred length format.

6.4 Special Architectures for ASN.1 Encoders / Decoders

The speed of software-based ASN.1 EDs is always limited by the speed of the computer where encoding / decoding as well as other protocol functionalities are implemented. On the other hand, a special-purpose architecture optimized for PDU encoding / decoding and parallel to the host machine can provide much faster encoding / decoding service [Bil 92].

The special-purpose ASN.1 encoding / decoding architectures are examples of the PDU encoding / decoding architectures described in chapter 4. These architectures are collection(s) of the distributed implementation model IMP, which is given in chapter 3; and of the phase algorithms for PDU encoding / decoding, which are introduced in chapter 2. In all of these architectures, ASN.1 encoding / decoding is done in a way similar to the ISODE approach. The major difference is that transformations between different forms of PDU values are done concurrently and different transformations are performed in parallel in separate IMPs.

The use of IMP as the main building block of different architectural models for ASN.1 EDs facilitates a modular design process. According to the design process, first components of IMP, such as the common processor model which is used as CC, IC, and EUs, as well as queues, and arbiters are designed. Based on the collection of these components an IMP is constructed. The software component for processors with different functionalities for all the algorithms are designed accordingly. In this section, first the hardware component of IMP is introduced. Based on the hardware component, the details of the code for the implementation of the phase algorithms on IMP are discussed. Then, different architectural models using IMPs for ASN.1 encoding / decoding are explained.

6.4.1 Hardware Component of IMP

As introduced in chapter 3, IMP consists of processors with different functionalities, namely CC, IC and EUs. In order to reach a regular design, a single type of processor is used for all these units. The use of a single type of processor for different functional units necessitates its programmability. The programmability of its processors also makes an ASN.1 ED adaptable to the changes in the encoding rules, or the host environment. Based on this programmability, the same architecture can be used for ASN.1 encoding / decoding based on different encoding rules [Bil 90].

Another important requirement in the design of IMP is to minimize the complexity of the components. This is necessary to maximize the number of EUs which can be put into a single VLSI chip along with CC and IC. The use of a Reduced Instruction Set Computer (RISC) as the basic building block of IMP becomes the only choice to satisfy the above requirements.

All the characteristics of the RISC machines, e.g. fewer instructions, fixed instruction format, single machine cycle for most of the instructions are desired characteristics of the processor implementing different units of IMP. Behavioural simulations of IMP has

shown that a small set of instructions is adequate to implement the phase algorithms for PDU encoding / decoding.

The RISC designed to implement the different units of IMP is a 16–bit processor with 16 operational registers. The instruction set includes a total of 16 instructions which are described in Table 6.3. All instructions have a fixed width of one word for simplicity and efficiency of the instruction fetch and sequence mechanisms.

The basic clock rate of the RISC is selected to be 8 MHz which is divided into four subcycles. A four stage pipeline is used such that all of the instructions except store, load, jump, call, and return take 1–cycle.

Table 6.3 Instruction Set of the RISC for IMP

Instruction	Format	Description
add	add Rd , Rs1 , Rs2	(Rd) = (Rs1) + (Rs2)
sub	sub Rd , Rs1 , Rs2	(Rd) = (Rs1) - (Rs2)
and	and Rd , Rs1 , Rs2	(Rd) = (Rs1) & (Rs2)
or	or Rd , Rs1 , Rs2	(Rd) = (Rs1) || (Rs2)
shr	shr Rd , Rs , sha	sha : shift amount (4 bit)
shl	shl Rd , Rs , sha	
ldh	ldh Rd , imm	imm : immediate (8 bit)
ldl	ldl Rd , imm	
ld	ld Rd , Rb , disp	Rb : base register
sto	sto Rs , Rb , disp	disp : displacement (4 bit)
call	call label	
ret	ret	
jmp	jmp cc , label	cc : condition code (4 bit)
gsw	gsw Rd	(Rd) = (status word)
ssw	ssw Rs	(status word) = (Rs)
nop	nop	

The RISC for IMP is a 16–bit machine, since a 64 Kbyte of local memory space is shown to be more than enough for all functional units. In order to perform the external memory operations, a virtual address scheme is used. The most significant four bits of

the status register of the RISC are reserved as the *memory mode identifier*. In this way, the virtual memory space addressed by the RISC working as IC becomes 2^{20} words.

According to the refinement 3.4, CC and IC communicate with EUs through buffers dedicated to each EU and queues which store the message requests coming from EUs. The buffers dedicated for EUs, in other words I/O registers for EUs are 2 banks of 8 16–bit registers. Both controllers and EUs access the I/O registers via memory mapped input (ld) and output (sto). The first register of either register bank is used to show the status of the register bank, as full, or empty. The other registers are used to store the contents of messages transferred between CC and EUs and IC and EUs. The possibility of conflict between controllers and EUs while writing into and reading from these registers is solved through a fixed priority scheme where read and write cycles of controllers precede those of EUs.

The inputs of the controller queues are connected to the trigger outputs of EUs. The trigger signals for CC and IC are generated after the last word of the outgoing message and is written to the I/O register by the EU. The details of the message passing software are described in the next subsection. The controllers access their respective queues via memory mapped I/O. The controller queues consist of eight 8–bit registers used to store unique identifier patterns generated for each trigger source. Similar to a software queue, two pointers implemented as counters are used to control the operation of the controller queues.

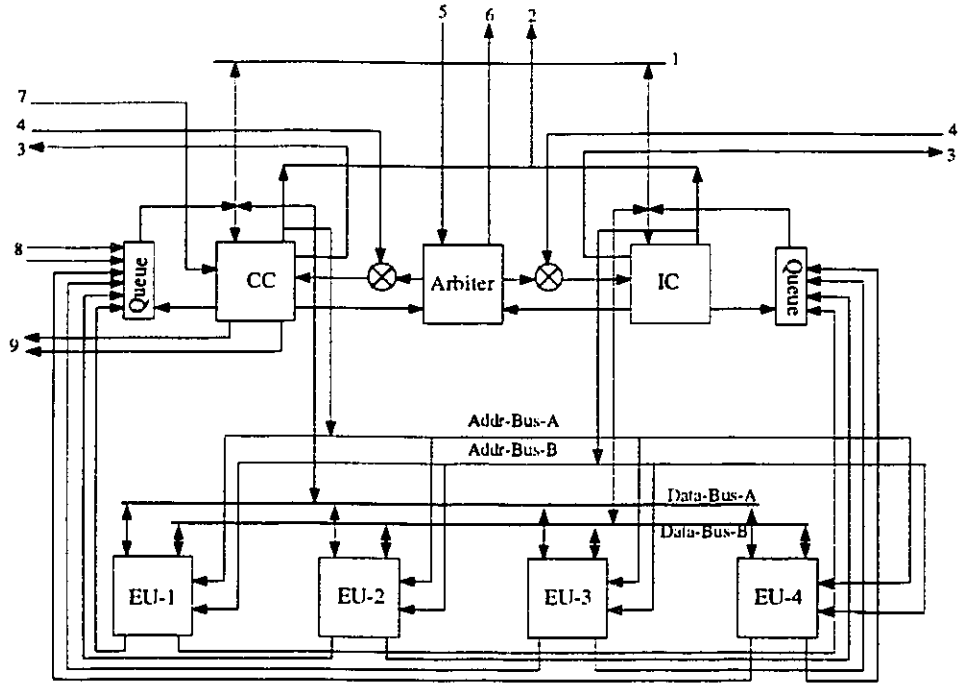
In order to resolve the conflicts between various processors on the shared resources both from within and from outside the modules, such as common memory, and module port, an arbiter mechanism is needed. The arbiters use a rotating priority scheme which is implemented based on a circular counter and a latch bank. The arbiters respond to requests coming from N different sources by a single acceptance and N-1 rejections.

In order to handle the case of a processor requesting more than one resource with the possibility of contention, an arbiter result decoder is also used in the design. The arbiter result decoder returns an acceptance response to the requesting processor if and only if all the responses from different resources are acceptance. Otherwise, a rejection response is sent to the requesting processor.

In Fig. 6.15, the configuration for IMP with four EUs is shown. This particular configuration shows the external connections of IMP for the (PM_d)-(M_eA) architectural model. According to the model, both address and data ports of the module are shared between CC and IC. Similarly, external memory interfaces are shared between two modules. This sharing is provided by the help of rotating prority arbiter mechanism. As an example, we can analyze the communication between CCs of two different modules.

When CC of a module wants to write into the I/O register file of the other CC, it sends a request to the arbiter of the other module through one of the lines in external connection-3, and another request to the arbiter of its module. The response sent from other module is transmitted through one of the lines in external connection-4 to the arbiter result decoder which also receives the reponse from the local arbiter. When both of the responses are acceptance, the decoder grants the connection to CC. More detailed information about the hardware component of IMP is given in [Bil 91].

Figure 6.15 IMP Configuration with 4 EUs for (PM_d)-(M_eA) Model



⊗ : Arbiter result decoder

1 : Data <15:0>
2 : Address <19:0>
3 : Local Lock Request <5:0>
4 : Local Lock Response <5:0>
5 : External Lock Request
6 : External Lock Response
7 : CC - I/O address
8 : Interrupts <1:0> (incoming)
9 : Interrupts <1:0> (outgoing)

6.4.2 Software Component of IMP

The software to implement the phase algorithms for ASN.1 encoding and decoding according to the BER and the DER is based on the common distributed implementation model specified in section 3.2.

As explained in the previous subsection, EUs communicate with controllers through their I/O registers. Two separate banks of 8 16–bit registers are used for the communication with CC and IC. The first register in both banks is used as the status register for the I/O bank; therefore the maximum message size is limited to seven 16–bit words. Since, there are different types of messages for each phase algorithm a *message identifier*, which

consists of the identity of the phase algorithm and the identity of the message, is used to distinguish the messages exchanged between processors.

In Fig. 6.16, the program segment to send a message from an EU to IC, to write the attributes of an intermediate value node, is shown. The first component of the message is the *message identifier*, followed by the intermediate value node's attributes, *identifier*, *length*, and *contents*. In order to set the status of the I/O registers to *to be read*, EU puts the word FFFF into the first I/O register. An interrupt, generated when the status is changed, is sent from EU to the IC queue. In order to block the proceeding of EU until the message is read by IC, EU polls the status register.

Figure 6.16 EU Code To Send a *WriteIVNd* Message to CC

```
      ldh Rf , 08;       /* Rf : base register for message transfer */
      ldl  Rf , 00;      /* (Rf) <= 0800 */
      ..............
      sto R6 , Rf , 9;   /* M(0809) <= 0002 - message type : write the intermediate value node */
      sto R3 , Rf , A;   /* M(080A) <= identifier for the intermediate value node */
      sto R4 , Rf , B;   /* M(080B) <= length for the intermediate value node */
      sto R5 , Rf , C;   /* M(080C) <= contents pointer for the intermediate value node */
      sto Rd , Rf , 8;   /* M(0808) <= FFFF - set the status of I/O to to be read */
wait : ld  R6 , Rf , 8;  /* R6 <= M(0808) - read the status of I/O */
      sub R6 , R6 , Rd;  /* check the status of I/O */
      jmp eq , wait;     /* if the status of I/O is still to be read , then wait till it becomes read */
      .....
```

Fig. 6.17 illustrates the portions of the program segment to receive a *WriteIVNd* message sent from an EU by IC. Initially, IC checks whether or not its request queue is full. When there is a request, and once its source is solved, IC reads the *message identifier* to determine the type of the message. When the message type is found to be *WriteIVNd*, IC reads all the intermediate value attributes first, and then sets the status of the I/O registers of the source EU as *empty*.

Figure 6.17 IC Code To Receive a *WriteIVNd* Message from an EU

```
        ldh Rf , 10;        /* Rf : base register for message transfer */
        ldl  Rf , 00;       /* (Rf) <= 1000 */
        ..............
        ldl Rf , 40;        /* Rf <= 1040 */
nomsg : ld  R2 , Rf , 0;    /* R2 <= queue word */
        and R2 , R2 , R1;   /* R1 : mask to check the source of requests */
        jmp ze  , nomsg;    /* if zero then there is no request source waiting to be served */
        ..............      /* code to decide the source of the request */
  eu1 : ldl Rf , 00;        /* Rf <= 1000 assuming the source is EU1 */
        ..............      /* code to set masks */
        ld  R2 , Rf , 9;    /* R2 <= message type  */
        ..............      /* code to compare the message type with defined patterns */
        ld  R3 , Rf , A;    /* R3 <= identifier for the intermediate value node */
        ld  R4 , Rf , B;    /* R4 <= length for the intermediate value node */
        ld  R5 , Rf , C;    /* R5 <= contents pointer for the intermediate value node */
        sto R0 , Rf , 8;    /* M(1008) <= 0000 - set the status of I/O to empty  */
        ..............
```

The message transfer in the reverse direction, i.e. from controllers to EUs is similar to the method described above. The only difference is that instead of using a separate interrupt line going from controllers to each EU, EUs simply poll their respective I/O registers which are written by the controllers.

Another important aspect of the software component of the IMP for ASN.1 encoding / decoding is the possibility of processing multiple PDUs concurrently. This technique is especially useful when processed or generated trees are flat, i.e. the set of nodes or functional units to be processed at the next step always contains a single element. In order to facilitate processing multiple PDUs concurrently, a *PDU identifier* is attached to each message sent between processors of IMP. The processing of multiple PDUs concurrently does not add any additional burden on either EUs or IC. However, the code for CC must be modified to store information about different PDUs at the same time and switch from one to another based on the messages received from EUs.

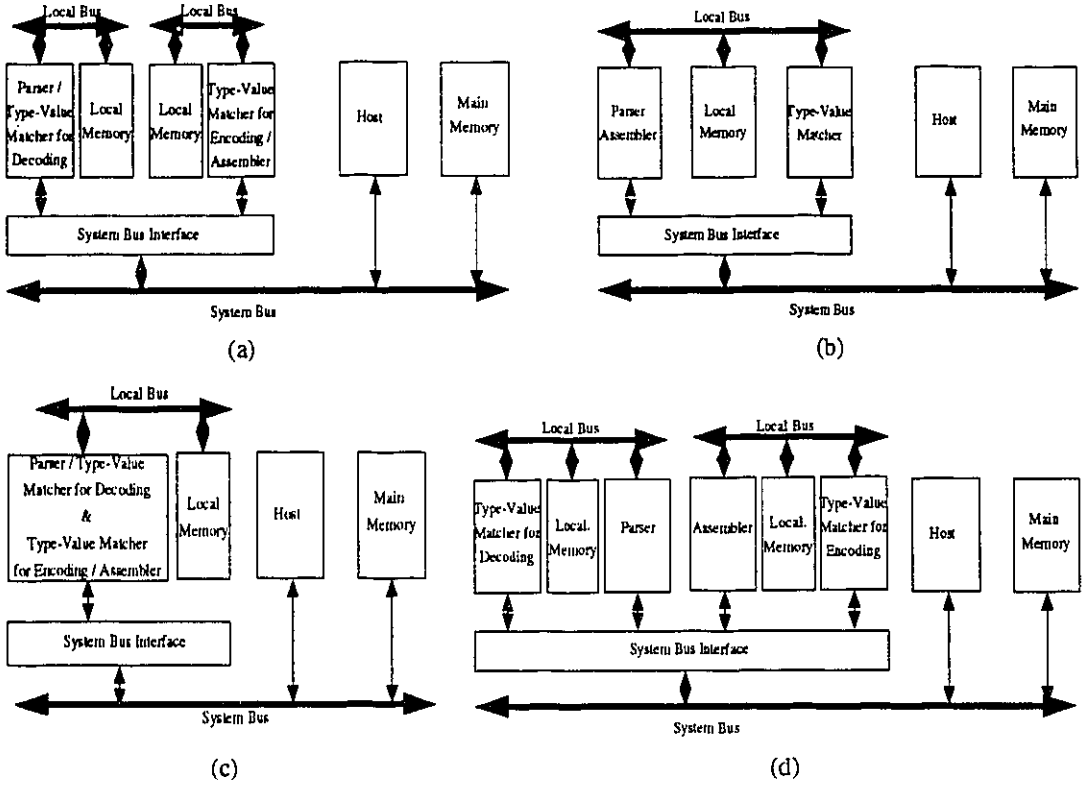
6.4.3 Architectures

As explained in chapter 4, there are various architectural models combining IMPs to obtain PDU EDs. Similarly, an ASN.1 ED can be obtained from different architectural configurations of the IMP for ASN.1 encoding / decoding which are explained in the subsections 6.4.1 and 6.4.2.

While constructing the ASN.1 EDs from IMPs, the important issue to be addressed is the communication among IMPs when they are used in architectures other than (PM_dM_eA), and the communication between IMPs and the host machine where other protocol functionalities are implemented.

Single ASN.1 EDs based on all four architectural models defined in chapter 4 are shown in Fig. 6.18. In all models, IMPs communicate with the host and the main memory through the system bus of the host. In order to access the system bus, an interface is used in all of the models. In (PM_dM_eA) and (PM_d)-(M_eA) models, the local memory modules, which are used to store the intermediate values, are accessed by a single IMP. On the other hand, in (PA)-(M_dM_e) and (P)-(M_d)-(M_e)-(A) models, the local memory modules are shared by two IMPs performing two phases of ASN.1 encoding and decoding according to BER or DER. In all models, local buses connect IMPs to local memory modules as well as to the other IMP (if any exists) which shares the local memory module. In Fig. 6.18, IMPs are shown with two connections, one to a local bus and the other to the system bus interface. However, in our implementation, in order to reduce the I/O complexity, IMPs have only one address and one data port to communicate. Therefore, the ports for each IMP should be shared between different sources of communication which are CC and IC of the IMP, and CC of the connected IMP.

Figure 6.18 ASN.1 Encoder / Decoder based on (a) (PM_d)-(M_eA) Model, (b) (PA)-(M_dM_e) Model, (c) (PM_dM_eA) Model, (d) (P)-(M_d)-(M_e)-(A) Model



Message passing mechanisms between communicating IMPs, and between IMPs and the system bus interface are implemented on CCs of IMPs. On the other hand, local memory modules are only accessed by ICs of associated IMPs. As can be seen from Fig. 6.18, there is no contention on the local bus for the models (PM_d)-(M_eA) and (PM_dM_eA); since in these models IMPs do not communicate with other IMPs. Instead IC reads from and writes to the local memory through the local bus. However, as in the other models, there is a possibility of contention between CC and IC for the I/O ports of the IMP when multiple PDUs are being encoded or decoded concurrently, as explained in the previous subsection. CC of IMPs of these models communicate with the host only to receive an encoding or decoding request and to send an encoding or decoding response. On the other hand, IC of IMPs accesses the main memory during the phases of encoding and decoding extensively. In order to solve the conflicts while accessing the I/O ports of

IMP in the case of possible conflicts, both CC and IC first try to obtain the local lock which is granted by the arbiter of IMP.

In the models (PA)-(M_dM_e) and (P)-(M_d)-(M_e)-(A), there is also the possibility of contention on the local bus between CCs and ICs of communicating IMPs. In order to solve the possible conflicts, each processor attempting to send a message or to access local memory, first tries to obtain two locks, one local and the other lock either in the other IMP or in front of the local memory. When a processor obtains two locks, it gets the local bus and sends its message.

In order to send a message, i.e. an encoding or decoding response to the system bus interface, CCs of IMPs write into registers provided in the system bus interface. When the system bus interface wants to send a message, which is written in its registers, it sends an interrupt to the queue of CC. Message passing between CCs of IMPs is similar; when CC of an IMP tries to send a message to the CC of another IMP it first writes the message, i.e. the first phase transformation response, into the I/O registers of the CC of another IMP, and then sends an interrupt to the queue of CC.

6.5 Performance Evaluation

In chapter 5, a performance evaluation methodology is proposed to measure the effectiveness of the special-purpose components of multilayered communication architectures. According to the methodology, performance figures of components of communication architectures are obtained by using techniques such as simulation, and benchmarking or by thorough estimations. In this section, we discuss the results of two studies :

• the performance comparisons between a ASN1 EDs based on multiple-layer single ED models and two software based EDs.

• the measurements for the effectiveness of different architectural models for ASN.1 encoding / decoding.

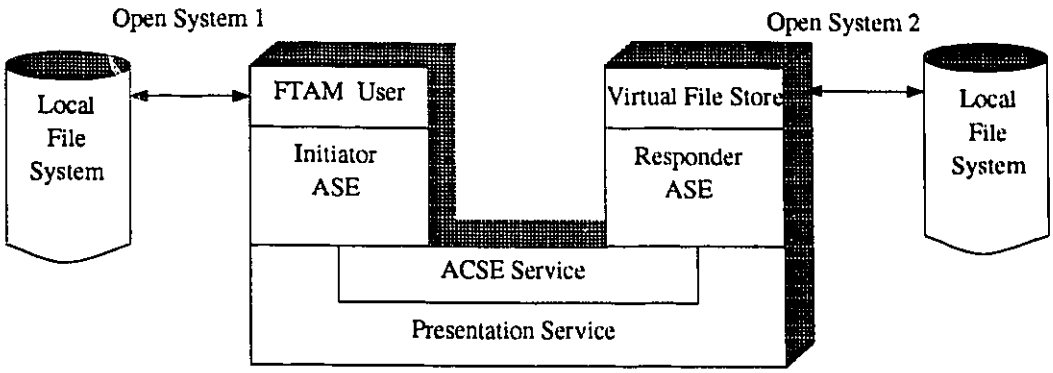
6.5.1 Performance Comparisons of ASN.1 Encoders / Decoders

In order to make comparisons between different EDs, first a set of PDUs must be selected. In some performance studies of ASN.1 EDs, they are tested using imaginary PDUs whose intermediate value trees have different forms and different sizes. The number of levels of value trees and the number of nodes at a level can be used as measurement parameters [Bil 90, Nak 88]. However, in order to obtain performance figures which can be used while evaluating the effectiveness of the architecture, PDUs of the real communication must be used.

In order to obtain performance comparisons between VASN1 and software based ASN.1 EDs, the PDUs transferred during the use of OSI File Transfer Access Management (FTAM) [ISO 8571–4] service of ISODE are used. These PDUs include FTAM PDUs, OSI Association Control Service Element (ACSE) [ISO 8650–1] PDUs and Presentation PDUs which are used to carry these Application PDUs. The structure of the protocol stack formed for FTAM connections is shown in Fig. 6.19.

FTAM is an application layer protocol for transferring, accessing, and managing files between open systems. The FTAM protocol is connection oriented with a series of embedded regimes which are, FTAM association, file selection, file open, and data transfer. During the FTAM association regime, i.e. for connection establishment and release, the FTAM user and the virtual file store are connected to the ACSE entities which use the presentation service. In other regimes, the FTAM user and the virtual file store directly communicate using the presentation service.

Figure 6.19 Structure of Protocol Stack for FTAM



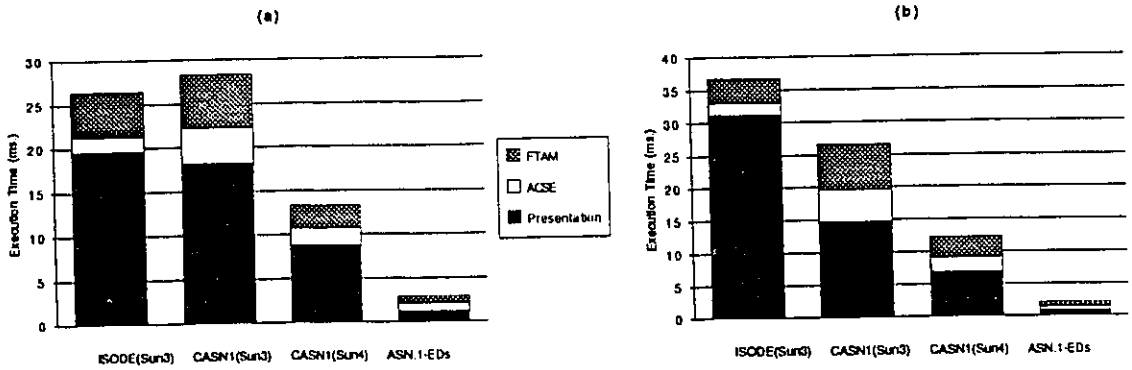
The software based ASN.1 EDs used in the performance comparison tests are the library of ISODE software distribution package, release 6.0 and encoding / decoding routines generated by the CASN1 compiler. The measurements for both ISODE and CASN1 are done on a Sun 3/60 running in a single user mode. The measurements for CASN1 are also performed on a Sun 4 Sparc 2 machine, again running in a single user mode. In order to measure the encoding / decoding time for various PDUs of routines, the set of application and presentation PDUs which are sent between two open systems during an FTAM session are used. The encoding / decoding time figures are the mean of 10 measurements. In each measurement, the routine tested is repeated 1000 times. In order to obtain the execution time, the UNIX™ *getitimer* facility is used.

The performance figures for different architectural models of ASN.1 EDs are obtained by simulation. The simulation of architectures are done on their structural models which are defined using the Very High Speed Integrated Circuits (VHSIC) Hardware Design Language (VHDL) [IEEE 1076]. VHDL is a hardware description language which can be used to specify a given design in various levels of abstraction - ranging from algorithmic level to silicon level [Arm 89]. VHDL is also used in the development phase of the design for IMP [Wu 90]. The structural models for four different multiple-layer single

EDs are simulated on the same set of PDUs used to evaluate the software based EDs. In all models each IMP includes four EUs.

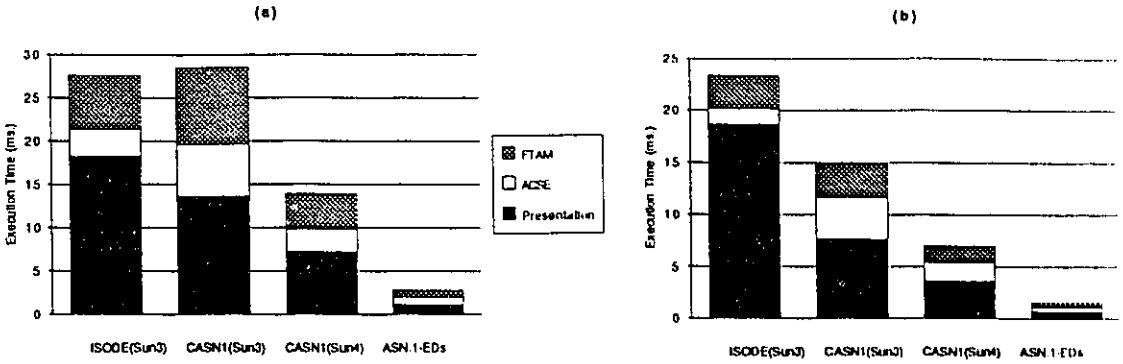
The set of PDUs used in measurements are divided into subsets for each FTAM regime. In Fig. 6.20.(a)-(b), the total encoding time and the total decoding time figures for FTAM, ACSE and Presentation PDUs associated with the FTAM-connection-request are respectively given. In these measurements, the encoding / decoding times for each PDU are separately obtained. Since the pipelining is not considered, the measurements for all four multiple-layer single EDs are the same. The encoding / decoding times are obtained as the sum of execution times for two phases in each case. The bar graphs include the measurement performed for multiple-layer single EDs, two measurements for routines obtained by using CASN1, and one measurement for the ISODE library routines. The bar graphs show the total encoding and decoding time measurements in terms of encoding / decoding time figures for FTAM, ACSE, and Presentation PDUs.

Figure 6.20 Measurements for PDUs Exchanged During
FTAM-Connection-Request : (a) Encoding Time, (b) Decoding Time



The encoding / decoding time measurements, performed for Application and Presentation PDUs sent from the open system where the virtual file server resides to the open system of the FTAM user, are shown in Fig. 6.21.(a)-(b).

Figure 6.21 Measurements for PDUs Exchanged During
FTAM-Connection-Response : (a) Encoding Time, (b) Decoding Time

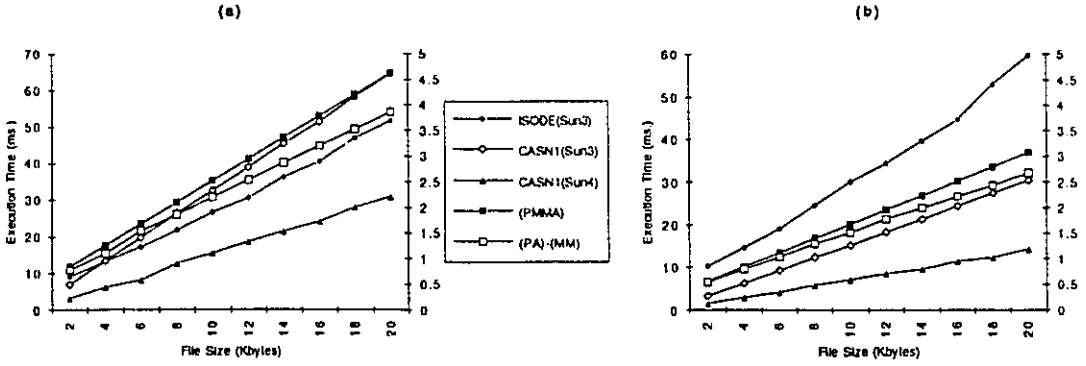


In both Fig. 6.20 and Fig. 6.21, the total execution times for phases of encoding and decoding on multiple-layer single ED architectures are much smaller than those for encoding / decoding routines of ISODE and CASN1. In all cases, most of the execution time is spent for processing Presentation PDUs. This is more pronounced in the case of ISODE, since the conversions between presentation elements and presentation streams are part of the Presentation PDU processing.

Another set of measurements are conducted in the data transfer regime. The ISODE FTAM service supports unstructured text, unstructured binary, and filedirectory files for the purpose of transmission. In order to measure the encoding / decoding time figures for bulk data transfer, the unstructured text file is used. In the bulk data transfer phase, the FTAM initiator entity residing at the same open system with the FTAM user separates the file into pieces to be written at the virtual file store. Similarly, the FTAM responder entity residing at the same open system with the virtual file store separates the file into pieces to be read by the FTAM user. Both initiator and responder use the presentation service to transfer the PDUs which include the file pieces. In Fig. 6.22.(a)-(b), the total encoding and decoding times for bulk data PDUs to transfer various sizes of files are shown. Since, the execution times for ASN.1 EDs are significantly lower than those for software based EDs, two different scales are used to chart the results. The y-axis on the

left of the graphs is used for ISODE, and CASN1 encoding / decoding routines. The y-axis on the right of the figures is for ASN.1 EDs. Since a number of PDUs are being sent together to be encoded or decoded, we can observe the effect of pipelining in the performance for the (PA)-(M_dM_e) and (P)-(M_d)-(M_e)-(A) models which yield the same encoding and decoding times. On the other hand, there is no possibility of pipelining in the (PM_dM_eA) and (PM_d)-(M_eA) models, and the encoding and decoding times are the sum of execution times of phase transformations. In these evaluations, each IMP processes at most one PDU at any given instance. In Fig. 6.22 and Fig. 6.23, (PMMA) is used to denote the (PM_dM_eA) and (PM_d)-(M_eA) models, whereas (PA)-(MM) denotes the (PA)-(M_dM_e) and (P)-(M_d)-(M_e)-(A) models.

Figure 6.22 Measurements for Bulk Data PDUs : (a) Encoding Time, (b) Decoding Time

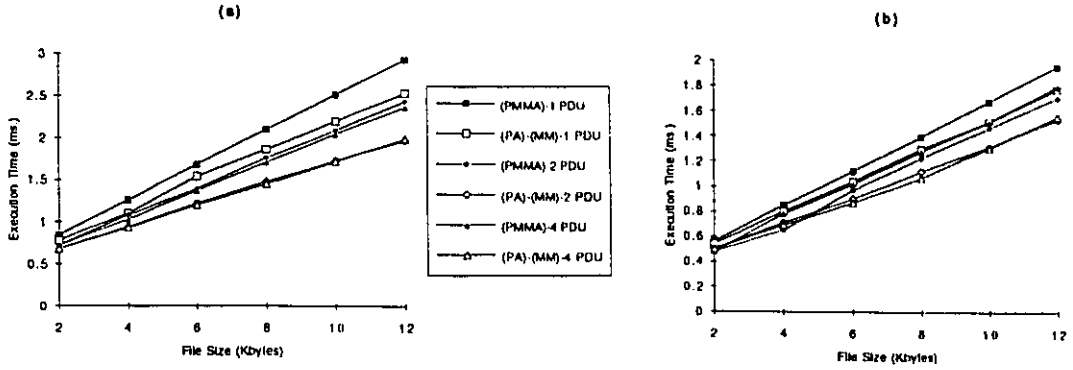


According to Fig. 6.22, for both encoding and decoding of bulk data PDUs, which are used to transfer unstructured text files, the performance of multiple-layer single ED architectures are much superior to the performances of encoding / decoding routines of ISODE and CASN1. This superiority is much pronounced when the encoding / decoding time figures for the (PA)-(M_dM_e) and (P)-(M_d)-(M_e)-(A) models for larger file sizes are considered. This is simply due to the limited use of the pipeline of these models.

Another test is done to compare the performance of software based EDs with those of the multiple-layer single EDs when multiple PDUs are concurrently processed in an

IMP. Performance figures are obtained for architectures for three different limits for a maximum number of PDUs which can be concurrently processed in an IMP. The total encoding and decoding times for PDUs of different file sizes are shown in Fig. 6.23.(a)-(b), respectively.

Figure 6.23 Measurements for Bulk Data PDUs When IMPs Process Multiple PDUs : (a) Encoding Time, (b) Decoding Time



According to Fig. 6.23, for all models, when multiple PDUs are processed concurrently a small reduction in the execution times are observed. However, there is almost no change between the case when two PDUs and the other case when four PDUs can be processed concurrently in an IMP. In some cases, even an increase in the execution time is observed when the limit increases from two to four. Although these results change with the structure and the size of the PDU as well as the number of EUs in IMPs, it demonstrates that when too many PDUs are processed concurrently in the same module, the queueing delays for **request** messages coming from EUs affects the overall execution time substantially.

Until now, we discussed the results of simulations for four different ASN.1 encoding / decoding architectures and compared them with the performance figures obtained for the ISODE library routines and the encoding / decoding routines generated by using CASN1. However, in order to measure the effectiveness of the architectures, we should also consider the effects of placing such EDs in a layered communication architecture.

6.5.2 Measuring the Effectiveness of ASN.1 Encoders / Decoders

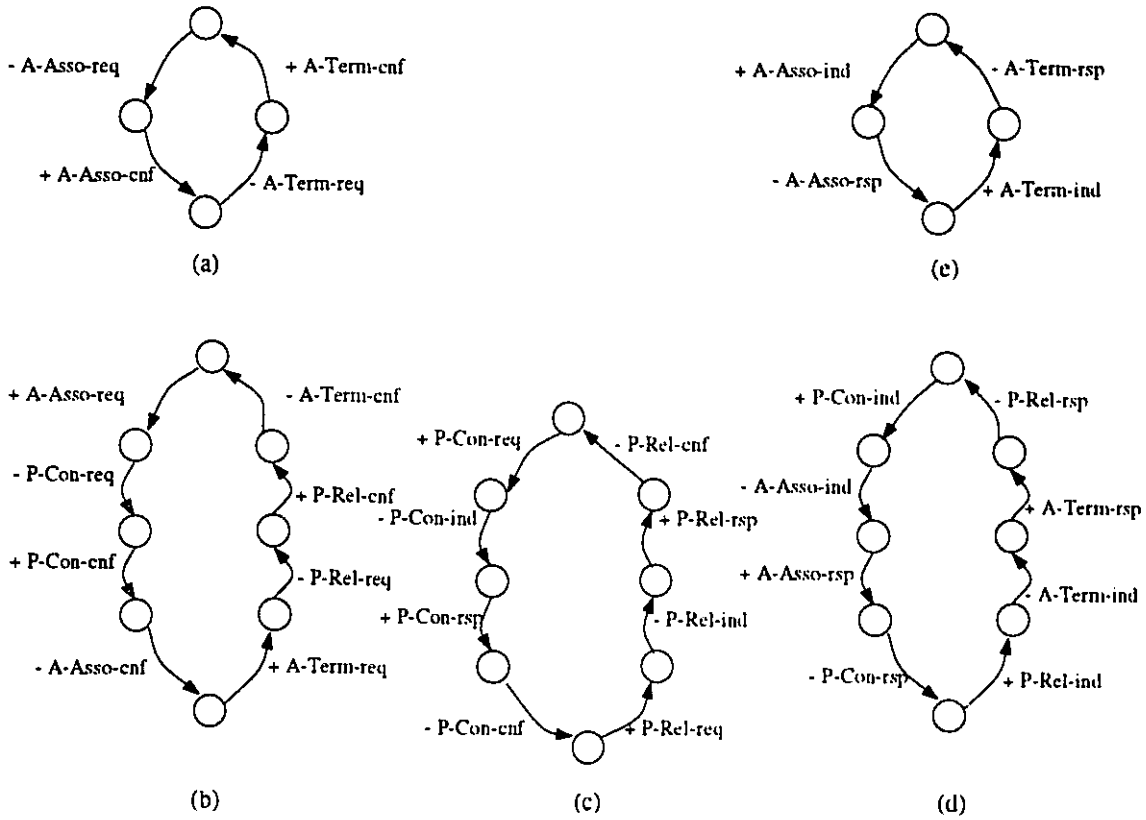
In order to measure the effectiveness of different ASN.1 encoding / decoding architectures, the methodology introduced in chapter 5 is used. The end-to-end performance for an OSI application, i.e. FTAM is evaluated when different models of ASN.1 EDs are used.

The application, presentation, and session layers of the tested protocol stack is implemented by the ISODE software package, release 6.0, on a Sun 3/50 and a Sun 3/60 machine. The Sun3/50 machine is used as the responder site where the virtual file store is situated. The Sun3/60 machine acts as the initiator site which serves to the FTAM user. The transport layer in the ISODE package is based on the implementation of OSI transport layer class 0 on top of the TCP/IP stack. Both workstations are connected to the same 10 Mb Ethernet-based local area network.

According to the terminology of chapter 5, layer L_{hdi} is the application layer, whereas layer L_{ldi} becomes the presentation layer since only these layers utilize an ASN.1 ED. Therefore, while the application and the presentation layer become DI layers, all the layers below are II layers, according to definition 5.2.3.

In order to obtain the $ATRG_{hdi,ldi}$, simplified forms of layer CFSMs are used. As an example Fig. 6.24 shows the CFSMs for ACSE. In order to simplify the resulting transition-relation graphs, certain functionalities of ACSE, i.e. association rejection, and association abort are excluded from the CFSMs.

Figure 6.24 CFSMs for ACSE : (a) UI, (b) EI, (c) US, (d) ER, (e) UR



Algorithm 5.2 is used to obtain the $ATRG_{hdi,ldi}$ which combines the transition-relation graphs for FTAM, ACSE and the presentation layer generated by Algorithm 5.1.

Once the $ATRG_{hdi,ldi}$ is obtained for DI layers, the next step is to obtain the $DTRG_{hdi,ldi}$ according to the structure graphs for processing overheads associated with each state of the $ATRG_{hdi,ldi}$. Different structure graphs for processing overheads which include PDU encoding and decoding are generated for each encoding / decoding architecture. In these graphs all the other tasks are assumed to be carried out on host machines implementing two open systems where the FTAM user and the virtual file server reside. When ISODE's library routines are used for ASN.1 encoding / decoding, encoding and decoding tasks are also performed on the host machines. The mean service time for

tasks performed on the host machines are obtained as the mean of 10 measurements. In each measurement, tasks are repeated 1000 times. In order to obtain the execution time, the UNIX™ *getitimer* facility is used. The measurements performed in the previous subsection are used as the mean service time for tasks associated with Application and Presentation PDU encoding / decoding.

The queueing network for the DI layers consist of two PS queues, which model the host machines, and for FCFS queues which model the IMPs of the ASN.1 EDs.

In order to construct the queueing network for the II layers, a simplified form of the methodology given in [Con 88] is used. Instead of constructing the model starting from the physical layer, a performance study for the measurement of overhead in TCP/IP stacks [Cla 89] is used. The mean service time figures for tasks associated with the transport layer and the session layer are obtained using the UNIX™ *getitimer* facility. The processing queues for the II layers are two PS queues modeling the host machines, and the FCFS queue which models the TCP/IP stack.

The end-to-end performance of FTAM service is evaluated in terms of total connection time needed to transfer a given number of a given size of unstructured text files. The connection time is the period between the instance a job of class FTAM-connection-request enters the queue, which models the host machine at the initiator site, and the instance when the same job leaves the same queue as class FTAM-connection-termination-confirm. The evaluations are done by varying following parameters :

- Number of transferred files during a connection (N_{rw}),
- FTAM service request rate (λ) in terms of requests per second (req/sec),
- Probability of read requests in total file transfer requests (P_r).

In Fig. 6.25.(a), the speed-up obtained when an ASN.1 encoding / decoding

architecture is used instead of the software routines run on the host machine, are shown for different FTAM service request rates. Fig. 6.25.(b) shows the effectiveness of four architectures for varying λ. In this evaluation set, a file read and a file write request are equally probable, and on average, one file is transferred during each connection. Measurements are done on files of size 10000 bytes. In this evaluation set and in the following other sets, the limit for the maximum number of PDUs which can be processed concurrently in the same IMP is set to 2.

Figure 6.25 Changing FTAM Service Request Rate : (a) Speed-up, (b) Effectiveness

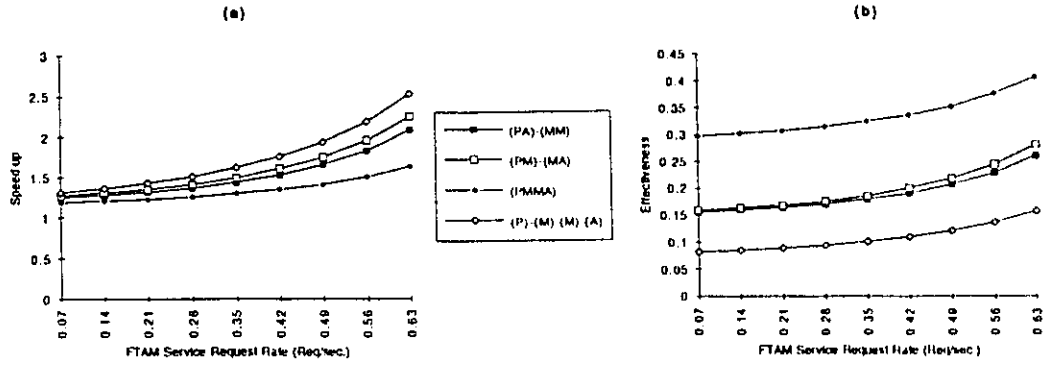
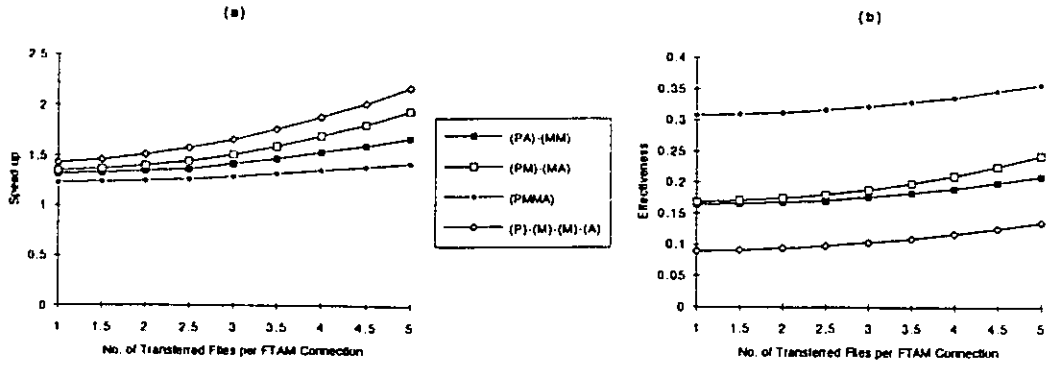


Fig. 6.25.(a) shows the maximum speed-up is obtained when an (P)-(M_d)-(M_e)-(A) model ASN.1 ED is used. It also shows that the speed-up increases with the increasing λ. According to Fig. 6.25.(b), the most effective encoding / decoding architecture under the given conditions is the simplest model, (PM_dM_eA). Fig. 6.25.(b) shows that the given load does not necessitate a highly-parallel encoding / decoding architecture. This is due to the high performance overhead associated with the FTAM service.

Fig. 6.26.(a)-(b), shows the speed-up due to the use of ASN.1 encoding / decoding architectures and their effectiveness for a varying number of file transfers per each connection. Similar to the previous evaluation set, a file read and a file write request are equally probable and the average FTAM service request rate is 0.21 req/sec. Measurements are done on files of size 10000 bytes.

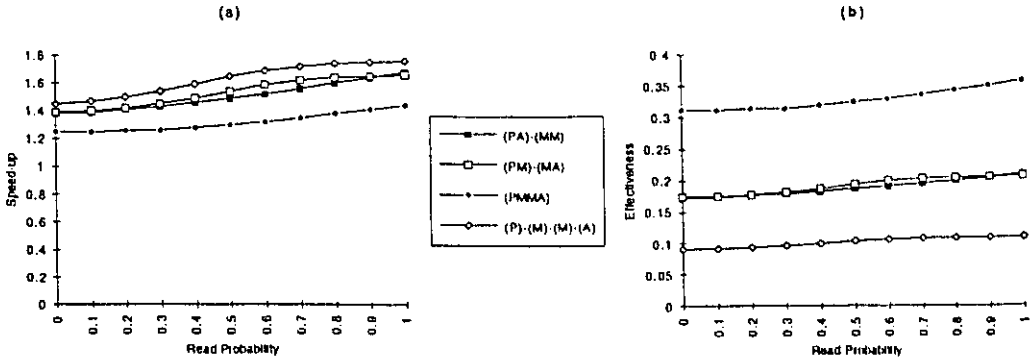
Figure 6.26 Changing Number of File Transfers per Connection : (a) Speed-up, (b) Effectiveness



As in the previous evaluation set, an increasing speed-up is observed due to the use of ASN.1 encoding / decoding architectures parallel to hosts with the increasing number of transferred files per FTAM connection. Similar to the previous case, due to the much higerher processing overheads associated with the FTAM service compared to those for ASN.1 encoding / decoding, the most effective encoding / decoding architecture is the (PM_dM_eA) model.

Fig. 6.27.(a)-(b), shows the speed-up due to the use of ASN.1 encoding / decoding architectures and their effectiveness while changing the probability of read requests in total file transfer requests. In this evaluation set, on the average, two files are transferred during each connection and the average FTAM service request rate is 0.35 req/sec. Measurements are done on files of size 10000 bytes.

Figure 6.27 Changing Probability of Read Requests : (a) Speed-up, (b) Effectiveness



According to Fig. 6.27.(a), the speed-up increases when a greater percentage of the transferred files are sent from the virtual file store to the FTAM user. This increase is mainly due to the fact that greater number of PDUs are needed to control the writing of a file at the virtual file store than reading it. Fig. 6.27.(a) also shows that when the data transfer becomes more one directional, either read or write, the architectures exploiting the bidirectional parallelism, i.e. (PM_d)-(M_eA) and (P)-(M_d)-(M_e)-(A) yield lower increases in the speed-up.

Chapter 7 CONCLUSIONS AND FUTURE WORK

In this chapter, the conclusions reached after the thesis study are presented and an outline of the possible future works in this area are introduced.

7.1 Conclusions

In this thesis, different aspects of PDU encoding / decoding in communication architectures are addressed. The main motivations in this study are threefold : to formalize different notions relating to PDU encoding and decoding; to construct concurrent algorithms and architectures for PDU encoding and decoding based on this formalization; and to measure the effect of PDU encoding / decoding architectures on the end-to-end performance of communication architectures.

In chapter 1, the PDU encoding / decoding functionality is introduced as a part of layered communication architectures. Similarities between PDU processing and value passing mechanisms in heterogeneous environments are discussed.

The framework for PDU encoding / decoding is outlined in chapter 2. The notions of PDU type and PDU value, as well as the existing duality between these notions are formalized. Different forms of PDU values, that is, local, intermediate, and global values, are described and they are then used to define PDU encoding and decoding as transformations amongst these forms. The encoding and decoding transformations are shown to be divisible into phases when certain constraints are satisfied. The phase transformations are further defined in terms of different forms of PDU values and their respective types. The phase transformations between local and intermediate values are

defined as type-value matching for encoding and decoding. The phase transformations between global and intermediate values are defined as assembling and parsing for encoding and decoding, respectively. Parallel algorithms for these phase transformations are described using functions which define the transformation for units of different PDU value forms.

The common distributed implementation model - IMP - for the parallel phase algorithms is explained in chapter 3. The IMP is described using the CSP notation; since in the design of the IMP, message passing is used as the method of communication between components, the CSP notation is found to be the most suitable for describing the IMP. The final IMP is developed as a result of a series of refinements which are performed to reflect different features of the IMP, such as synchrony, failure, and message buffering. The safety and liveness properties of the parallel phase algorithms implemented on IMP are shown. The message complexity for the type-value matching algorithms on IMP is found to be $O\left(\sqrt[k]{MN}\right)$ for a well-balanced global type tree of size M and a local or an intermediate value tree of size N where the global type tree includes type trees for all PDUs of a given protocol and both trees have the height k. The best-case time complexity for the type-value matching algorithms on IMP with $O\left(\frac{N}{log\ N}\right)$ processors is found to be $O\left(\sqrt[k]{M}\ log\ N\right)$. The message complexity for parsing and assembling algorithms on IMP is found to be $O(N)$ for an intermediate value tree or a global value of size N. The best-case time complexity for the type-value matching algorithms on IMP with $O\left(\frac{N}{log\ N}\right)$ processors is found to be $O(log\ N)$.

Different configurations for the IMP, which are based on which phase algorithms are implemented, are defined in chapter 4. Four different single PDU encoding / decoding architectures are constructed using these different configurations of the IMP model. Single PDU encoding / decoding architectures serving multiple layers and multiple sources are developed from the original four architectures. Multiple PDU encoding / decoding

architectures consisting of single PDU EDs are further defined.

In chapter 5, a performance evaluation methodology to measure the effectiveness of different single and multiple PDU encoding / decoding architectures as parts of layered communication architectures is proposed. The effectiveness of a PDU ED is defined as the increase in the end-to-end performance of the communication architecture resulting due to the use of the PDU ED with respect to the total number of processors. The performance evaluation methodology is based on the partition of layers of the communication architecture with respect to their use of the PDU ED and the construction of queueing models for these partitions using different techniques. These queueing models are then combined together and solved using an iterative algorithm.

The overall applicability of the concepts presented in the previous chapters is demonstrated through the development, implementation, and performance evaluation of ASN.1 encoding / decoding architectures used in OSI implementations, is presented in chapter 6.

In this chapter, the results of performance comparisons of various ASN.1 encoding / decoding architectures with software based EDs, and the measurements of the effectiveness of these architectures when placed in an OSI-based communication architecture are presented. The comparisons yield that for all encoding / decoding architectures, the execution times are substantially lower with respect to the execution times for software based EDs. This is found to be more pronounced when the pipelining of architectures is utilized. The measurements of the effectiveness of PDU encoding / decoding architectures show that with the increasing number, size, and complexity of PDUs, the architectures become more effective on the improvement of the end-to-end performance of communication architectures.

7.2 Future Work

As outlined in the first chapter, due to the lack of a standard language to specify PDUs of different layers, and of different protocols, the formalism for PDUs should depend on other methods of representation, such as trees, and strings. However, such a language, if ever designed, would greatly affect the communication architecture implementations. First, it may be used to further refine the notions introduced in chapter 2. It may also serve as the input for generating the software component of PDU encoding / decoding architectures which can be used for different layers, and different protocols.

In chapter 6, ASN.1 encoding / decoding is used as the example application to test the concepts developed in this thesis. Similar PDU EDs can also be built for processing PDUs of lower layers of the OSI reference model, as well as data units exchanged in other communication architectures, and implementing value passing mechanisms in heterogeneous environments. It should be noted that although the design presented for ASN.1 encoding / decoding can be used with the proper software component for other layer PDUs, implementations using simpler components can be developed for lower layer PDUs. Therefore, the most suitable design should be constructed according to the complexity of PDUs to be processed using the abstract IMP.

Similar to the comparison studies and effectiveness measurements performed for ASN.1 encoding / decoding architectures by using FTAM, further tests can be carried out on different OSI applications. Another future work is to investigate the effects of the use of large number of EUs in IMPs. Due to the limitations of available tools, simulations are done for architectures including at most 24 processors. Although, the most optimum configuration in terms of the speed up in the encoding and decoding times is highly PDU dependent, a study investigating the effects of larger numbers of EUs in IMPs may indeed be valuable.

REFERENCES

[Aho 72] A.V. Aho, and J.D. Ullman, "Theory of Parsing, Translation, and Compiling : Volume 1 : Parsing," Prentice-Hall, 1972.

[Aho 83] A.V. Aho, J. E. Hopcroft, and J.D. Ullman, "Data Structures and Algorithms," Addison-Wesley, 1983.

[Arm 89] J. R. Armstrong, "Chip-level Modeling with VHDL," Prentice-Hall, 1989.

[Bas 75] F.Baskett, K.M. Chandy, R.R. Muntz, and F. Palacios, "Open, Closed and Mixed Networks of Queues with Different Classes of Customers," J. ACM, vol. 22, no. 2, pp. 248–260, 1975.

[Bir 84] A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," ACM Trans. Comput. Syst., vol. 2, no. 1, pp. 39–59, 1984.

[Bil 90] M. Bilgic and B. Sarikaya, "An ASN.1 Encoder / Decoder and Its Performance," 10th International IFIP WG 6.1 Symposium on Protocol Specification, Testing, and Verification, L. Logrippo, R. Probert, and H. Ural (Eds.), New York, Elseiver, pp. 133–152, 1990.

[Bil 91] M. Bilgic, W. Wu, and B. Sarikaya, "VASN1–A High-Speed Protocol Encoder / Decoder," submitted for publication, IEEE Trans. Commun., 1991.

[Bil 92] M. Bilgic and B. Sarikaya, "Performance Comparison of ASN.1 Encoder / Decoders Using FTAM," Computer Communication, (accepted for publication), 1992.

[Bla 87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald," IEEE Trans. Soft. Eng., vol. 13, no. 1, pp. 65–76, 1987.

[Bla 91] U. Black, "OSI – A Model for Computer Communications Standards," Prentice-Hall, 1991.

[Bra 90] F. Brady, A.G. Boshier, D. Pitt, B.M. Szcygiel, "One2One – A Tool for Translating ASN.1 to ACT ONE," Proceedings of the 3rd International Conference on Formal Description Techniques, 1990.

[Bro 91] S.D. Brookes and A.W. Roscoe, "Deadlock Analysis in Networks of Communicating Processes," Distributed Computing, vol. 4, no. 4, pp. 209–230, 1991.

[Cla 89] D.D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead," IEEE Commun. Mag., vol. 27, no. 6, pp. 23–29, 1989.

[Con 88] A.E. Conway, "A Generic Performance Model for Multi-layered OSI Communication Architectures," GTE Lab. Inc., Waltham, MA, Tech. Memo. TM-0069–10–88–414.17, 1988.

[Day 83] J.D. Day and H. Zimmermann, "The OSI Reference Model," Proc. IEEE, vol. 71, no. 12, pp. 1334–1340, 1983.

[Dij 80] E.W. Dijkstra and C.S. Scholten, "Termination Detection for Diffusing Computations," Inform. Process. Lett., vol. 11, no. 1, pp. 1–4, 1980.

[Dub 90] M. Dubiner, Z. Galil, and E. Magen, "Faster Tree Pattern Matching," 31st Annual Symposium on Foundations of Computer Science (FOCS'90), pp. 72–86, 1990.

[Fdi 90] S. Fdida, H.G. Perros and A. Wilk, "Semaphore Queues : Modeling Multilayered Window Flow Control Mechanisms," IEEE Trans. Commun., vol. 38, no. 3, pp. 309–317, 1990.

[Fei 78] E. Feinler and J. Postel, *Arpanet Protocol Handbook*, Network Information Center, SRI International, Menlo Park, Calif., 1978.

[Gau 89] P. Gaudette, S. Trus, and S. Collins, "An Object–Oriented Model for ASN.1," Proceedings of the First International Conference on Formal Description Techniques, pp.121–134, 1989.

[Gib 87] P.B. Gibbons, "A Stub Generator for Multilanguage RPC in Heterogeneous Environments," IEEE Trans. Soft. Eng., vol. 13, no. 1, pp. 77–87, 1987.

[Gih 86] O. Gihr and P.J. Kuehn, "Comparison of Communication Services with Connection-Oriented and Connectionless Data Transmission," Computer Networking and Performance Evaluation, T. Hasegawa, H. Takagi, and Y. Takahashi (Eds.), New York, Elseiver, pp. 173–186, 1986.

[Gun 89] P. Gunningberg, M. Bjorkman, E. Nordmark, S. Pink, P. Sjodin, and J.E. Stromquist, "Application Protocols and Performance Benchmarks," IEEE Commun. Mag., vol. 27, no. 6, pp. 30–36, 1989.

[Hec 91] E. Heck, D. Hogrefe, and B. Muller-Clostermann, "Hierarchical Performance Evaluation Based on Formally Specified Communication Protocols," IEEE Trans. Comput., vol. 40, no. 4, pp. 500–513, 1991.

[Her 82] M. Herlihy and B. Liskov, "A Value Transmission Method for Abstract Data Types," ACM Trans. Program. Lang. Syst., vol. 4, no. 4, pp. 527–551, 1982.

[Hoa 78] C.A.R. Hoare, "Communicating Sequential Processes," Comm. ACM, vol. 21, no. 8, pp. 666–677, 1978.

[Hoa 81] C.A.R. Hoare, "A Calculus of Total Correctness for Communicating Processes," Sci. Comput. Programming, vol. 1, no. 1–2, pp. 49–72, 1981.

[Hoa 85] C.A.R. Hoare, "Communicating Sequential Processes," Prentice-Hall, 1985.

[Hof 82] C.M. Hoffmann and M.J. O'Donnell, "Pattern Matching in Trees," J. ACM, vol. 29, no. 1, pp. 68–95, 1982.

[IEEE 1076] "IEEE Standard VHDL Language Reference Manual," Institute of Electrical and Electronics Engineers, IEEE Std 1076–1987.

[ISO 3309] "Information Processing Systems-Open Systems Interconnection-High–Level Data Link Control (HDLC) –Frame Structure," International Organization for Standardization, International Standard ISO 3309.

[ISO 8073] "Information Processing Systems-Open Systems Interconnection-Connection Oriented Transport Protocol Specification," International Organization for Standardization, International Standard ISO 8073.

[ISO 8571–4] "Information Processing Systems-Open Systems Interconnection-File Transfer, Access, and Management (FTAM), Part 4 : The File Protocol Specification," International Organization for Standardization, International Standard ISO 8571–4.

[ISO 8650–1] "Information Processing Systems-Open Systems Interconnection-Protocol Specification for the Association Control Service Element (ACSE)," International Organization for Standardization, International Standard ISO 8650.

[ISO 8824] "Information Processing Systems-Open Systems Interconnection-Specification of Abstract Syntax Notation One (ASN.1)," International Organization for Standardization, International Standard ISO 8824.

[ISO 8825] "Information Processing Systems-Open Systems Interconnection-Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)," International Organization for Standardization, International Standard ISO 8825.

[ISO 8825–2] "Information Processing Systems-Open Systems Interconnection-Committee Draft-Specification of ASN.1 Encoding Rules-Part 2 : Packed Encoding Rules," International Organization for Standardization, Committee Draft CD-8825–2.

[ISO 8825–3] "Information Processing Systems-Open Systems Interconnection-Committee Draft-Specification of ASN.1 Encoding Rules-Part 3 : Distinguished Encoding Rules," International Organization for Standardization, Committee Draft CD-8825–3.

[ISO 9072] "Information Processing Systems-Open Systems Interconnection-Remote Operations, Part 2 Protocol Specification," International Organization for Standardization, International Standard ISO 9072.

[ISO 9594–5] "Information Processing Systems-Open Systems Interconnection-The Directory, Part 5 : Access and System Protocols Specification," International Organization for Standardization, International Standard ISO 9594–5.

[Jif 89] H. Jifeng, "Various Simulations and Refinements," Proceedings on Stepwise Refinement of Distributed Systems models, formalisms, correctness, Lecture notes in Computer Science, No. 430, J.W. de Bakker, W.P. de Roever, and G. Rozenburg(Eds.), pp. 340–360, 1989.

[Knu 77] D. Knuth, J. Morris, and V. Pratt, "Fast Pattern Matching in Strings," SIAM J. Comput., vol. 6, no. 2, pp. 323–350, 1977.

[Kos 89] S.R. Kosaraju, "Efficient Tree Pattern Matching," 30th Annual Symposium on Foundations of Computer Science (FOCS 89), pp. 178–183, 1989.

[Kri 86] P.S. Kritzinger, "A Performance Model of the OSI Communication Architecture," IEEE Trans. Commun., vol. 34, no. 6, pp. 554–563, 1986.

[Kri 87] A.S. Krishnakumar, B. Krishnamurthy, K.K. Sabnani, "Translation of Formal Protocol Specifications to VLSI Designs," 7th International IFIP WG 6.1 Symposium on Protocol Specification, Testing, and Verification, H. Rudin and C.H. West (Eds.), New York, Elseiver, pp. 375–390, 1987.

[Kue 86] P.J. Kuehn, "Modeling of New Services in Computer and Communication Networks," Computer Networking and Performance Evaluation, T. Hasegawa, H. Takagi, and Y. Takahashi (Eds.), New York, Elseiver, pp. 283–303, 1986.

[Lam 76] S.S. Lam, "Store-and-Forward Buffer Requirements in a Packet Switching Network," IEEE Trans. Commun., vol. 24, no. 4, pp. 394–403, 1976.

[Lam 77] L. Lamport, "Proving the Correctness of Multiprocess Programs," IEEE Trans. Soft. Eng., vol. 3, no. 2, pp. 125–143, 1977.

[Lam 90] L. Lamport and N. Lynch, "Distributed Computing : Models and Methods," Chapter 18 in Formal Models and Semantics, Handbook of Theoretical Computer Science, vol. B, J.V. Leeuwen (Ed.), pp. 1157–1199, Elseiver, 1990.

[Lav 83] S.S. Lavenberg (Ed.), "Computer Performance Modeling Handbook, New York, Academic, 1983.

[Lis 88] B. Liskov, "Distributed Programming in Argus," Commun. ACM, vol. 31, no. 3, pp. 300–312, 1988.

[Mei 87] A. Meijer, "Systems Network Architecture," Pitman, 1987.

[Mit 86] L.C. Mitchell and D.A. Lide, "End-to-end Performance Modeling of Local Area Networks," IEEE J. Select. Areas Commun., vol. 4, no. 6, pp. 975–985, 1986.

[Mol 82] M.K. Molloy, "Performance Analysis Using Stochastic Petri Nets," IEEE Trans. Comput., vol. 31, no. 9, pp. 913–917, 1982.

[Mur 88] M. Murata and H. Takagi, "Two-layer Modeling for Local Area Networks," IEEE Trans. Commun., vol. 36, no. 9, pp. 1022–1034, 1988.

[Nak 88] T. Nakawaji, K. Katsuyama, N. Miyauchi, and T. Mizuno, "Development and Evaluation of APRICOT(tools for abstract syntax notation one)," Proc. 2nd Int. Symp. Interoperable Inform. Syst., Tokyo, Japan, pp. 55–62, Nov. 1988.

[Neu 90] G.W. Neufeld and Y. Yang, "The Design and Implementation of an ASN.1–C Compiler," IEEE Trans. Soft. Eng., vol. 16, no. 10, pp. 1209–1220, 1990.

[Not 88] D. Notkin, A.P. Black, E.D. Lazowska, H.M. Levy, J. Sanislo, and J. Zahorjan, "Interconnecting Heterogeneous Computer Systems," Comm. ACM, vol. 31, no. 3, pp. 258–273, 1988.

[Pos 81] J.B. Postel, C.A. Sunshine, and D. Cohen, "The ARPA Internet Protocol," Computer Networks, vol. 5, no. 4, pp. 261–271, 1981.

[Ree 88] J. Reed and R.T. Yeh, "Specification and Verification of Liveness Properties of Cyclic, Concurrent Processes," ACM Trans. Program. Lang. Syst., vol. 10, no. 1, pp. 156–177, 1988.

[Rei 82] M. Reiser, "Performance Evaluation of Data Communication Systems," Proc. IEEE, vol. 70, no. 2, pp. 171–196, 1982.

[Rei 86] M. Reiser, "Communication-System Models Embedded in the OSI Reference Model : A Survey," Computer Networking and Performance Evaluation, T. Hasegawa, H. Takagi, and Y. Takahashi (Eds.), New York, Elseiver, pp. 85–111, 1986.

[Ros 87] A. Roscoe, N. Dathi, "The Pursuit of Deadlock Freedom," Inf. Comp., vol. 75, no. 3, pp. 289–327, 1987.

[Rud 83] H. Rudin, "From Formal Protocol Specification Towards Automated Performance Prediction," 3rd International IFIP WG 6.1 Symposium on Protocol Specification, Testing, and Verification, H. Rudin and C.H. West (Eds.), New York, Elseiver, pp. 257–269, 1983.

[Rud 84] H. Rudin, "An Improved Algorithm for Estimating Protocol Performance," 4th International IFIP WG 6.1 Symposium on Protocol Specification, Testing, and Verification, Y. Yemini, R. Strom, and S. Yemini (Eds.), New York, Elseiver, pp. 515–525, 1985.

[Sal 85] J.H. Saltzer, D.D. Clark, J.L. Romkey, and W.C. Gramlich, "The Desktop Computer as a Network Participant," IEEE J. Select. Areas Commun., vol. 3, no. 3, pp. 468–478, 1985.

[Sha 84] A.U. Shankar, S.S. Lam, "Specification and Verification of Time-Dependent Communication Protocols," International IFIP WG 6.1 Symposium on Protocol Specification, Testing, and Verification, Y. Yemini, R. Strom, and S. Yemini (Eds.), New York, Elseiver, pp. 215–226, 1985.

[Sou 83] N. Soundararajan, "Correctness Proofs of CSP Programs," Theor. Comput. Sci., vol. 24, pp. 131–141, 1983.

[Sun 85] Sun Microsystems, *External Data Representation Reference Manual*, Sun Microsystems, 1985.

[Svo 89] L. Svobodova, "Implementing OSI Systems," IEEE J. Select. Areas Commun., vol. 7, no. 7, pp. 1115–1130, 1989.

[Tan 88] A.S. Tanenbaum,"Computer Networks," Prentice–Hall, 2nd ed., 1988.

[Wes 78] C.H. West, "General Technique for Communications Protocol Validation," IBM J. Res. Develop., vol. 22, no. 4, pp. 393–404, 1978.

[Whi 89] J.E. White, "ASN.1 and ROS : The Impact of X.400 on OSI," IEEE J. Select. Areas Commun., vol.7, no. 7, pp. 1060–1072, 1989.

[Wol 88] Wollongong Group, *ISODE: The ISO Development Environment*, User Manual, Wollongong Group, Palo Alto, Ca., 1988.

[Wu 90] W. Wu, M. Bilgic, and B. Sarikaya, "VHDL Modeling and Synthesis of an ASN.1 Encoder / Decoder," CCVLSI-90, Ottawa, Oct. 1990.

[Xer 81] Xerox Corporation, *Courier: The Remote Procedure Call Protocol*, Tech. Rep. XSIS 038112, Xerox Corporation, 1981.

[You 67] D.H. Younger, "Recognition and Parsing of Context-Free Languages in Time n^3," Information and Control, vol. 10, pp. 189–208, 1967.

[Zha 89] K. Zhang and D. Shasha, "Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems," SIAM J. Comput., vol. 18, no. 6, pp. 1245–1262, 1989.